Refinement and Verification in Component-based Model Driven Design

Zhenbang Chen [a,d], Zhiming Liu [a,*] Anders P. Ravn [b]
Volker Stolz [a], Naijun Zhan [c]

[a] *UNU-IIST, P.O. Box 3058, Macao SAR, China*
[b] *Department of Computer Science, Aalborg University, Denmark*
[c] *Lab. of Computer Science, Institute of Software, CAS, Beijing, China*
[d] *National Laboratory for Parallel and Distributed Processing, Changsha, China*

Abstract

Modern software development is complex as it has to deal with many different and yet related aspects of applications. In practical software engineering this is now handled by a UML-like modelling approach in which different aspects are modelled by different notations. Component-based and object-oriented design techniques are found effective in the support of separation of correctness concerns of different aspects. These techniques are practised in a model driven development process in which models are constructed in each phase of the development. To ensure the correctness of the software system developed, all models constructed in each phase are *verifiable*. This requires that the modelling notations are formally defined and related in order to have tool support developed for the integration of sophisticated checkers, generators and transformations. This paper summarises our research on the method of Refinement of Component and Object Systems (rCOS) and illustrates it with experiences from the work on the Common Component Modelling Example (CoCoME). This gives evidence that the formal techniques developed in rCOS can be integrated into a model-driven development process and shows where it may be integrated in computer-aided software engineering (CASE) tools for adding formally supported checking, transformation and generation facilities.

Key words: Formal methods, multi-view modelling, rCOS, software design process, tool design, UML.

* Corresponding author: Zhiming Liu, UNU-IIST, P.O. Box 3058, Macao
Email address: z.liu@iist.unu.edu (Zhiming Liu).
URL: http://www.iist.unu.edu/~lzm/ (Zhiming Liu).

1 Introduction

Software engineering is now facing two major challenges:

- the rapidly increasing complexity of systems to be developed, and
- higher demands for correct quality software.

Thus we observe that software development is becoming complex due to many different but inter-related aspects or views of the system, including those of static structure, flow of control, interactions, and functionality, as well as issues about concurrency, distribution, mobility, security, timing, and so on. In addition to these problems in the design phases, the proper implementation of interactions among the GUI, the controllers of the hardware devices and the application software components is a demanding task.

An effective means to handle such complexity is *separation of concerns*; and assurance of correctness is enhanced by application of formal modelling and analysis.

Separation of concerns is to divide and conquer. At any stage of the development of a system, the system is divided according to the concerns of the different views. These views can be modelled separately and their integration forms a model of the whole system. Different concerns require different models and different techniques; the specification and design of the functionality can be done as a state-based static specification, while event-based techniques are the simplest for designing and analyzing interactions among different components. However, it is not easy to practice separation of concerns due to the following problems:

(1) the lack of a design process which supports the identification of the crucial aspects of the system at different stages of the development and the consistent design of the different aspects, and
(2) the lack of a semantic foundation to define the precise relations among the different views that effectively supports a coherent and comprehensive understanding of the system, and because of this
(3) the lack of integrated techniques and tools for consistent analysis and verification of the system. Analysis is needed, for example when one has to decompose a required "global" property into "local" properties of the different views.

The suggested approach in practical software engineering to dealing with software complexities is *component-based model-driven development* (CB-MDD). With such an approach,

- the different aspects and views of the system are described in a UML-like

multi-view and multi-notational language [43,49],
- separate design of the different aspects is supported by design patterns, object-oriented and component-based designs [32,20], and
- model construction is aided by UML-tools, design is supported by model transformation tools, and verification is supported mainly by testing tools.

In this approach, the Rational Unified Process (RUP) [30] is often adopted. The practical engineering methods with CB-MDD in a RUP help to some extent by providing visual notations for modelling *components* and their *provided and required interfaces*, the different aspects and views of components and their relations. This has greatly eased the difficulties in identification and building models of the different views of components and composition of components, with the support of powerful UML-modelling environments, such as MagicDraw [41]. However, to develop correct transformations and effective verification, there is need for a rigorous unified semantic theory and a coherent collection of high level theorems about the constructs, such that they support tool suites for specification, refinement and verification of the models. These are the main concerns of this paper, and the role that rCOS plays is to provide the formal definitions of the models in a UML-based RUP development process.

Rigorous specification, refinement and verification require the application of formal methods. In the past half a century, the formal method community has developed semantic theories, specification notations and tools of verification, including *static analysis*, *model checking*, *formal proof and theorem proving*, and *runtime checking*. They can be classified into the following frameworks:

- event-based models [39,25] are widely used for specification and verification of interactions, and they are supported by model checking and simulation tools [48,16];
- pre-post conditions and Hoare logic are applied to specifications of functionality and static analysis; these are supported by tools of theorem proving, runtime checking, static checkers and testing [34,18,38];
- automata, state transition systems, and temporal logics are popular for specification and verification of dynamic control behaviors; they are supported by model checking tools [28,33].

However, each framework is researched mostly by a separate community, and most of the research in verification has largely ignored the impact of design methods on feasibility of formal verification, automated verification in particular. Therefore, the verification techniques and tools are not very scalable and they are not easy to integrate into practical design processes.

The notion of *program refinement* has obvious links to the practical design of programs with the consideration of abstraction and correctness, and the well-

studied refinement calculi [40,3] are shown to be effective for the correctness of program level functionality; however there is a need for extension to object-oriented and component-based *model refinement* as witnessed by the recent work on formal methods of component and object systems [1,5,6,10,23,58].

In this paper, we present our ongoing research on a component-based design method, called rCOS. The method includes a modelling notation, directly defining concepts of *classes, objects, components, interfaces, contracts, coordination and composition*. The method is founded on a well-studied semantic model with a refinement calculus [23,10,58], which is the basis for formal verification and correct model transformation. We have also defined a UML profile for rCOS formal specifications, and refinement can be done on UML models using model transformations [15]. We discuss how this method can effectively support the integration of the techniques and tools of refinement and verification in a component-based model driven design process. In particular, we show in the CoCoME case study [12]:

(1) the system is specified by a collection of use cases, and each is modelled by an rCOS component;
(2) the models of different aspects of components at each stage of the development, including the *requirements elicitation and modelling*, *functionality design by refinement*, *logical component-based architecture design*, *detailed design and coding*, and *design of GUI and hardware controllers*;
(3) how these models are constructed using the UML profile and refined by application of design patterns that are proved to be refinements in rCOS;
(4) how verification and validation tasks are identified for the models and what are effective tools for these tasks.

Outline

In outline, the paper proceeds as follows. In Section 2 we give an overview of rCOS, showing how the significant concepts and artifacts of component-based software engineering are formalized in rCOS. This introduction tends to be informal and gives references to the literature where technical results are documented. This leads to a discussion in Section 3 about a software design process well integrated with rCOS. Section 4 is devoted to the rCOS development of the CoCoME case study, showing how rCOS is used for model construction and analysis with regard to the different design phases. The conclusions and related work are discussed in Section 5.

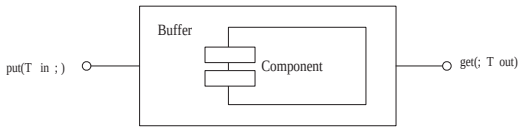


Fig. 1. A component with syntactic interface only

2 Introduction to rCOS

The research on rCOS develops a *model driven development method* that combines object-oriented and component-based design and analysis techniques. As a method, rCOS is founded on a *formal semantic theory* [23,10]; it includes a *modelling notation* with a calculus of *refinement* for object-oriented models and component models [23,10,58]; also it considers integration with a *development process*, from requirements elicitation through to coding. Within the process, the formal techniques and tools of modelling, design and verification can be applied, such as MasterCraft [52] for modelling and ModelMorf [36,13] for model transformation.

For these purposes, the rCOS semantic theory defines the important concepts and artifacts in the domain of object-oriented and component-based software engineering, like *classes, objects, components, interfaces, contracts*, *composition* (*connectors*), *coordination* and *glue*. It provides the behavioral semantics of these concepts with high level rules for refinement and verification. This section introduces the formalisation of these concepts in rCOS and they will be used in the later part of the paper for the design of the case study. Because of the limited space, we do not present formal syntax and only focus on the semantic formalisation of the software artifacts and their treatment. The syntax of rCOS is illustrated in the models of the case study.

2.1 Interfaces and contracts

An interface I provides the *syntactic type information* for an interaction point of a component. It consists of two parts: the *data declaration section*, *I.FDec*, that introduces a set of variables with their types, and the *method declaration section*, *I.MDec*, that defines a set of method signatures. Each signature is of the form $m(T_1\ in; T_2\ out)$, where T_1 and T_2 are type names, *in* stands for an input parameter, and *out* stands for an output parameter. We allow multiple input parameters and usually call the output parameter *return*.

Current practical component technologies provide syntactical aspects of interfaces only and leave the semantics to informal conventions and comments. This is obviously not enough for rigorous verification and validation. For example, the component with only syntactic interfaces shown in Fig. 1 has no

information about its functionality or behavior. For this, we define the notion of *contracts* of interfaces.

A *contract* is a specification of the semantics for the interface. However, different usages of the component in different applications have different needs:

- An interface contract for a component in a sequential system is obviously different from one in a communicating concurrent system. The former only needs to specify the functionality of the methods, e.g. in terms of their pre- and post-conditions, whereas the latter should include a description of the communication protocol, e.g. in terms of interaction traces, to specify the order in which the interactions happen.
- If the component is to be used in a real-time application, the contract of its interface must also specify real-time constraints, such as the lower and upper bounds of the execution time of a method.
- Components in distributed, mobile or internet-based systems require the contracts of their interfaces to include information about their locations.
- An interface (of a component) should be stateless when the component is required to be used dynamically and independently from other components. This means that the execution history of a component does not affect the other components.

Thus the rCOS framework takes great trouble to separate such concerns and viewpoints.

It is the contract of the interface that defines the external behavior and features of the component and allows the component to be used as a black box. Therefore in rCOS, we define *an interface contract for a component as a description of what is needed for the component to be used in building and maintaining software systems*. However, this description can be incremental in the sense that newly required viewpoints can be added when needed according to the application. Also, the consistency of these viewpoints should be formalizable and checkable. For this, rCOS builds on Hoare and He's Unifying Theories of Programming (UTP) [27].

2.2 A short introduction to UTP

In UTP, a *sequential program* (but possibly nondeterministic) is represented by a *design* $D = (\alpha, P)$, where

- α denotes the set of state variables (called observables). Each state variable comes in an unprimed and a primed version, denoting respectively a pre- and a post-state value. The set includes program variables, e.g. x, x', and a designated Boolean variable, ok, ok', that denotes termination or stability of

the program.
- P is of the form $p(x) \vdash R(x, x')$, called the functionality specification of the program. Its semantics is defined as $(ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$, meaning that if the program is activated in a stable state, ok, where the *precondition* $p(x)$ holds, the execution will terminate, ok', in a state where the postcondition R holds.

In UTP, the *refinement partial order* $\sqsubseteq$ among designs is defined such that $D_1 \sqsubseteq D_2$ if they have the same alphabet, say $\{x, x'\}$, and $\forall x, x' \cdot (P_2 \Rightarrow P_1)$ holds, where P_1 and P_2 are the functionality specification of D_1 and D_2, respectively. It is proven that with this order the set of designs forms a complete lattice, and *true* is the *least* (*worst*) element of the lattice. Furthermore, this lattice is closed under the classical programming constructs:

- *sequential composition*, $D_1; D_2$,
- *conditional choice*, $D_t \lhd g(x) \rhd D_f$, where g is a predicate, and D_t is selected when g evaluates to true, and D_f is selected when g evaluates to false.
- *nondeterministic choice*, $D_1 \vee D_2$, and
- *least fixed point of iterations*, $\mu x.D$.

All these constructs are *monotonic operations* on the lattice of designs. Refinement between designs is naturally defined as logical implications. These fundamental mathematical properties ensure that the domain of designs is a proper semantic domain for sequential programming languages. For a design, we define its *weakest precondition* for a given post condition q:

$$\mathbf{wp}(p \vdash R, q) \mathrel{\widehat{=}} p \wedge \neg(R; \neg q)$$

where the meaning of composing relations by ";" is the same as in UTP, $q_1; q_2 \mathrel{\widehat{=}} \exists v_0 \cdot (p_1[v_0/x'] \wedge p_2[v_0/x])$.

Semantics of *concurrent and reactive programs*, such as those specified by Back's action systems [2] or Lamport's Temporal Logic of Actions (TLA) [31], are introduced by the notion of *reactive designs* with an additional Boolean observable *wait* that denotes suspension of a program. A design P is a *reactive design* if it is a fixed point of $\mathcal{H}$, i.e. $\mathcal{H}(P) = P$, where

$$\mathcal{H}(p \vdash R) \mathrel{\widehat{=}} (wait \vee p) \vdash wait' \lhd wait \rhd R$$

We use a *guarded design* $g \& P$, where P is a design, to specify the reactive behaviour $\mathcal{H}(P) \lhd g \rhd (true \vdash wait')$, meaning that if the guard g is false, the program stays suspended, and if it is true, the result is $\mathcal{H}(P)$. The semantics of a reactive design is to ensure that a synchronisation of a method invocation by the environment and the execution of the method can only occur when the guard is true and *wait* is false.

The domain of reactive designs enjoys the same closure properties as that of sequential designs, and also refinement is defined as logical implication [24].

class C [**extends** D] {
attr $T\ x = d, \ldots, T\ x = d$
method $m(T\ in; V\ return)$ {
pre: $p \lor \ldots \lor p$
post: $\land\ (R; \ldots; R) \lor \ldots \lor (R; \ldots; R)$
$\land\ \ldots\ldots$
$\land\ (R; \ldots; R) \lor \ldots \lor (R; \ldots; R)$
}/* method m */
$\vdots$ $\vdots$
method $m(T\ in; V\ return)$ { $\ldots\ldots$ }
$\ldots\ldots$
invariant Inv }

Fig. 2. Specification of a class

2.3 Class structures and datatypes

Most, if not all, of the current component-based design and implementation techniques rely heavily on object-oriented techniques. For this, we allow the types of variables to be *classes*, and extend UTP with the notions of objects, classes, inheritance, polymorphism, and dynamic binding [23]. The design process starts with a requirements model that we write as a class specification in the format of Fig. 2.

An attribute is assumed to be public unless it is tagged with reserved words *private* or *protected*. The initial value of an attribute is optional; if no initial value of a class variable is declared, it will default to *null*.

The precondition of a method is a disjunction of simpler predicates that contain no primed variables, and the postcondition is a conjunction of disjunctions of relation composition of simple predicates that may contain both unprimed and primed variables. The reader can see the influence of TLA$^+$ [31], UNITY [9] and Java [21] on the above format.

A *design* $p \vdash R$ for a method is written as **Pre** p and **Post** R. An R in the postcondition is of the form $c \land (le' = e)$, where c is a condition, le an *assignable expression* and e an expression. An assignable expression le is either a variable x, an attribute name a or an indirect attribute name $le.a$. An expression e may be a logically specified expression such as the greatest common divisor of two given integers. We allow the use of *indexed conjunction* $\forall i \in I : R(i)$ and *indexed disjunction* $\exists i \in I : R(i)$ for a finite set I.

Also, specifications of classes at a lower level design or even program code are allowed by using, together with designs, the usual object-oriented programming primitive commands and constructs.

In rCOS, we distinguish data from objects and thus a datum, such as an integer or a boolean value does not have a reference. For this paper, we assume the elementary data types of

$$V ::= \textbf{long} \mid \textbf{double} \mid \textbf{char} \mid \textbf{string} \mid \textbf{boolean}.$$

The types T are then generated by:

$$T ::= V \mid C \mid \mathit{set}(T) \mid \mathit{bag}(T)$$

where C is a class name.

We use the operations $\mathit{add}(T\ a)$, $\mathit{contains}(T\ a)$, $\mathit{delete}(T\ a)$ on a set or a bag with their usual meaning. For a variable s of type $\mathit{set}(T)$, the specification statement $s.\mathit{add}(T\ a)$ equals $s' = s \cup \{a\}$, and $s.\mathit{contains}(T\ a)$ equals $a \in s$, and $s.\mathit{sumAll}()$ is the sum of all elements of s, which is assumed to be a set of numbers. We use curly brackets $\{e_1, \ldots, e_n\}$ and square brackets $[[e_1, \ldots, e_m]]$ to define a set respectively a bag. For a set s such that each element has an unique key or identifer, $s.\mathit{find}(\mathit{ID}\ \mathit{id})$ denotes the function that returns the element whose key equals id if there is one, or null otherwise. Notice that Java implements these types via the *Collection* interface. Therefore, these operations in specification statements can be easily coded in Java.

2.4 Contracts of interfaces

In the current version of rCOS, we consider components in concurrent and distributed systems, and a *contract* $\mathit{Ctr} = (I, Q, \mathcal{S}, \mathcal{P})$ of interface I specifies

- the allowable initial states by the *initial condition* Q, denoted by $\mathit{Ctr.Init}$,
- the specification function $\mathcal{S}$, denoted by $\mathit{Ctr.Spec}$, that assigns each method a guarded design $g\&D$, and
- the *interaction protocol* $\mathcal{P}$, denoted as $\mathit{Ctr.Prot}$, which is a set of sequences of call events, where a sequence is written: $?op_1(x_1), \ldots, ?op_k(x_k)$. Notice that a protocol can be specified by a temporal logic or a trace logic formula.

We use $\mathit{Ctr.IF}$ to denote the interface of contract Ctr.

Example 1 *The component interface in Fig. 1 does not say that the buffer is a one-place buffer. A one-place buffer can be specified by a contract* B_1 *for which*

- *The interface:* $B_1.\mathit{IF} = \langle q : \mathit{Seq}(\mathit{int}), \mathit{put}(\mathit{item} : \mathit{int};), \mathit{get}(; \mathit{res} : \mathit{int})\rangle$
- *The initial condition:* $B_1.\mathit{Init} = q = \langle\rangle$
- *The specification:*

$$B_1.\mathit{Spec}(\mathit{put}) = q = \langle\rangle \& \mathit{true} \vdash q' = \langle \mathit{item}\rangle$$

$$B_1.\mathit{Spec}(\mathit{get}) = q \neq \langle\rangle \& \mathit{true} \vdash \mathit{res}' = \mathit{head}(q) \wedge q' = \langle\rangle$$

- *The protocol:* $B_1.Prot$ *is a set of traces that is a subset of*

$$\{e_1, \ldots, e_k \mid e_i \text{ is ?put if } i \text{ is odd and ?get otherwise}\}.$$

The protocol corresponds to the semantics of a CSP process. However, instead of writing the CSP process, we use the rCOS tool to draw a sequence diagram in the UML profile of rCOS, that is then automatically translated into the CSP process. ■

2.5 Contract refinement

A contract Ctr has a denotational semantics (cf. [24]) in terms of its *failure set* $\mathcal{F}(Ctr)$ and divergence set $\mathcal{D}(Ctr)$, that is the same as the failure-divergence semantics for CSP [10]. Informally speaking, $\mathcal{D}(Ctr)$ contains of the *interaction traces* of the form $?m_1(u; v), \ldots ?m_n(u_n; v_n)$ that lead to *divergence*, i.e. $ok' = false$. $\mathcal{F}(Ctr)$ is the set of pairs (tr, F) where tr is an interaction trace and F is a set of method invocations such that the execution of tr leads to the refusal of the method invocations in F, i.e. their guards are falsified.

A contract Ctr_1 is *refined* by contract Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if the latter offers the same provided methods, $Ctr_1.IF.MDec = Ctr_2.IF.MDec$, is not more likely to diverge than the former, $\mathcal{D}(Ctr_1) \supseteq \mathcal{D}(Ctr_2)$, and not more likely to deadlock than the former, $\mathcal{F}(Ctr_1) \supseteq \mathcal{F}(Ctr_2)$. We have established in [24] a complete proof technique of refinement by simulation.

Theorem 1 (Refinement by Simulation) *$Ctr_1 \sqsubseteq Ctr_2$ if there exists a total mapping relating the state u of the attributes of Ctr_1 to the state v' of the attributes of Ctr_2, denoted $\rho(u, v') : Ctr_1.IF.MDec \longrightarrow Ctr_2.IF.MDec$, such that*

(1) $Ctr_2.Init \Rightarrow (Ctr_1.Init; \rho)$.
(2) $\rho \Rightarrow (guard_1(op) = guard_2(op))$ *for all* $op \in Ctr_1.IF.MDec$, *where* $guard_i(op)$ *denotes the guard of operation* op *in* Ctr_i.
(3) for each $op \in Ctr_1.IF.MDec$, $Ctr_1.Spec(op); \rho \sqsubseteq \rho; Ctr_2.Spec(op)$. ■

The need for the mapping to be total is to ensure that any state in the "abstract contract" is "implemented" in the refined contract. Similarly, contract refinement can also be proved by a surjective upward simulation [10].

Theorem 2 (Completeness of Simulations) *If $Ctr_1 \sqsubseteq Ctr_2$, there exists a contract Ctr such that*

$$Ctr_1 \preceq_{up} Ctr \preceq_{down} Ctr_2.$$

$\preceq_{up}$ and $\preceq_{down}$ denote upwards and downwards simulation, respectively. ■

2.6 Consistency

The formalization of contracts supports separation of views, but the different views have to be consistent. A contract *Ctr* is *consistent*, denoted by *Cons*(*Ctr*), if it will never enter a deadlock state when its environment interacts with it according to its protocol, i.e., if $\langle ?op_1(x_1), \ldots, ?op_k(x_k)\rangle \in Ctr.Prot$ then

$$\mathbf{wp}\left(\begin{array}{l} Init; g_1 \& D_1[x_1/in_1]; \ldots; g_k \& D_k[x_k/in_k], \\ \neg wait \wedge \exists op \in Ctr.IF.MDec \bullet guard(op) \end{array}\right) = true$$

Note that this formalization takes both synchronization conditions and functionality into account, as an execution of a method with its precondition falsified will diverge and a divergent state can cause deadlock too. We have proven the following *theorem of separation of concerns* [24]:

Theorem 3 (Separation of Concerns) *For* $i \in \{1, 2\}$

(1) If $Cons(I, Q, \mathcal{S}, \mathcal{P}_i)$*, then* $Cons(I, Q, \mathcal{S}, \mathcal{P}_1 \cup \mathcal{P}_2)$
(2) If $Cons(I, Q, \mathcal{S}, \mathcal{P}_1)$ *and* $\mathcal{P}_2 \subseteq \mathcal{P}_1$*, then* $Cons(I, Q, \mathcal{S}, \mathcal{P}_2)$
(3) If $Cons(I, Q, \mathcal{S}, \mathcal{P})$ *and* $\mathcal{S} \sqsubseteq \mathcal{S}_1$*, then* $Cons(I, Q, \mathcal{S}_1, \mathcal{P})$*.*

$\sqsubseteq$ *stands for the pointwise extension of the refinement relation over (guarded) designs to mapping functions.* ■

This allows us to refine the specification and the protocol separately.

2.7 Components

A component is an *implementation* of a contract. Formally speaking, a *component* is a tuple $C = (I, Q, MCode, PriMDec, PriMCode, InMDec)$, where

- I is an interface, called the *provided interface* and denoted by *C.pIF*,
- Q is an initialisation command, denoted by *C.Init*, setting the initial values of the attributes,
- *PriMDec*, denoted by *C.PriMDec*, is a set of method signatures, called *private methods* of the component,
- *MCode*, denoted by *C.MC*, and *PriMCode*, denoted by *C.PriC*, map a public method and a private method m respectively to a guarded command $g_m \rightarrow c_m$,
- *InMDec* is the set of required methods used by the code of the component, called the *required interface* and denoted by *C.rIF*.

The *semantics* $[\![C]\!]$ is a function that calculates a *general contract* for the provided interface from any given contract *InCtr* of the required interface

$$[\![C]\!](InCtr) \mathrel{\widehat{=}} ((I, Q, \mathcal{S}), PriMDec, PriMSpec)$$

It is a contract with *the specification of the private methods*, where the specifications $\mathcal{S}$ defines for each method m in the interface I a reactive design that is the semantics of the code $MCode(m)$ with the given specification of the required methods by *InCtr*. Similarly, *PriMSpec* defines for each private method a reactive design from its code [24].

A component C_1 is refined by another component C_2, denoted by $C_1 \sqsubseteq C_2$ if

(1) they provide the same methods, $C_1.pIF.MDec = C_2.pIF.MDec$
(2) they require the same methods $C_1.rIF = C_2.rIF$, and
(3) for any given contract of the required interface (called an *input contract*), the resulting provided contract of the latter refines that of the former, $C_1(InCtr) \sqsubseteq C_2(InCtr)$, for all input contracts *InCtr*.

Note that the notion of component refinement is useful for both *component correctness by design* and component *substitutability* in maintenance.

2.8 Simple connectors

To support the development activity, the semantic framework also needs to define operators for connecting components, resulting in new components, constructs for defining glue processes, and constructs for defining processes. In summary, the framework should be *compositional and support both functional and behavioral specification*. In rCOS, simple connectors between components are defined as component compositions. These include *plugging* (or *union*) and *service hiding* and *feedback*. These compositions are shown in Figs. 3-5. *Service renaming* is also provided so that names of methods in interfaces can be changed before components are composed.

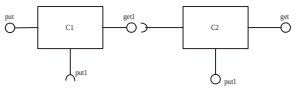


Fig. 3. Plugging

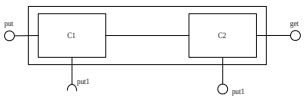


Fig. 4. Hiding

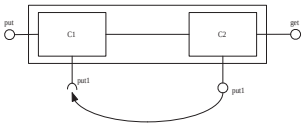


Fig. 5. Feedback

2.9 Coordination

Components provide a number of methods, but do not themselves activate the functionality specified in the contracts; we need active entities that implement

a desired functionality by coordinating sequences of method calls. These active entities are specially designed for a given application and therefore do not in general share the defining features of components.

To cater for this, we introduce in [10] *process components* into rCOS. Like a component, a process has an interface declaring its local state variables and methods, and its behavior is specified by a process contract. Unlike a component that is passively waiting for a client to call its methods, a process is active and has its own control on when to call out to required methods or to wait for a call to its provided methods. For simplicity, but without losing expressiveness [22], we assume a process does not provide methods and only calls methods provided by components. It is modelled by an interface and its associated contract (or code).

Let C be the component, that is formed by disjoint union of a number of disjoint components C_i, $i = 1 \ldots k$. A coordination program for C is a process P that calls a set X of provided methods of C. The composition $C \|_X P$ of C and P is defined similarly to the alphabetized parallel composition in CSP [48] with interleaving of events. The *coordination composition* is defined by hiding the synchronized methods (i.e. making them τ events) between the component C and the process P, that is $(C \|_X P) \backslash X$ and denoted as $(C \|_{[X]} P)$. Inspired by the work of Woodcock and Morgan [55], we have proven in [10] that $(C \|_{[X]} P)$ is a component. The coordination composition is illustrated in Fig. 6, where $C1$ and $C2$ are two one-place buffers and P is a process that keeps getting the item from $C1$ and putting it to $C2$. Therefore, processes only directly communicate via shared components. Of course, a component can also communicate indirectly with another component via processes, but without knowing the component with which it is communicating. Obviously, communication between a process and a component is by method invocation.

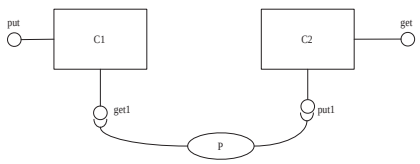


Fig. 6. Gluing two one-place buffers with a process forms a three-place buffer

An *application program* is a set of parallel processes that make use of the services provided by components. Analysis and verification of an application program can be performed in the classical formal frameworks, but at the level of contracts of components instead of implementations of components. The analysis and verification can reuse any proved properties about the components, such as divergence freedom and deadlock freedom without the need to reprove them.

3 Development Process with rCOS

With the rCOS models of the concepts and artifacts in component-based model-driven software engineering, we can accommodate a use-case driven, incremental and iterative Rational Unified Development Process [30]. Each iteration goes through the phases of *requirements elicitation and modelling*, *functionality design*, *logical component-based architecture design*, *detailed design and coding* and *design of GUI and hardware controllers*. Analysis and verification are carried out on the models produced in each of these phases.

3.1 Requirements elicitation and modelling

In the requirements elicitation and modelling, as what will be illustrated in Section 4.2, a number of *use cases* are captured, modelled and analyzed. Each use case is modelled in rCOS as a contract of an *interface* that a component provides to the *actors* of the use case. The *attributes* of the interface declare the domain objects involved in the realization of the use case. Further analysis on a set of use cases leads to the decomposition of some use cases into composition of use cases (cf. Section 4.2.3), and union of simple use cases into a composite use case (cf. Section 4.2.4, modelled by compositions of components through their interfaces.

The classes of these domain objects are models as a *conceptual class diagram*[1] in the UML profile of rCOS that can be translated to rCOS class declaration section, representing the structural view of the data and objects of the components. An rCOS class declaration section, see Section 4 for examples, is similar to a list of Java class declarations, but specification statements in the form of UTP designs are allowed [23].

The *interaction protocol* of the contract describes the interactions between the actors and the system for this use case. They are graphically represented as a sequence diagram called a *use case sequence diagram* in the UML profile, and translated to a CSP process in rCOS.

The flow of control and synchronization is modelled by a *state diagram* in the UML profile and the static functionality of interface methods is given in terms of their *pre* and *post conditions*. The combination of the state diagram and the static functionality specification is a *guarded design* for the contracts [11].

In summary, the initial formal model of the requirements consists of a set of

[1] This term is borrowed from [32]. The diagram is said to be conceptual, because the classes do not have any methods defined.

interfaces contracts in rCOS, one for each use case, each is to be designed into an rCOS components. The contracts should be analyzed to ensure their *consistency* and desirable properties such as safety and liveness. The contracts of the use cases form the *initial architecture* of the system.

3.2 Functionality design

This phase focuses on the *design* of the functionality of the use case methods specified in the contracts using the *OO refinement calculus* [23], component refinement, and substitution [10] in rCOS. This is mainly to refine the interactions between the actors and the system specified in the contracts of the use cases into interactions among the objects of the classes in the class diagram. There are three kinds of OO refinements involved: *functionality decomposition*, *attribute encapsulation*, and *class decomposition*.

3.2.1 Functionality decomposition

Assume that C and C_1 are classes, $C_1\ o$ is an attribute of C and $T\ x$ is an attribute of C_1. Let $m()\{c(o.x', o.x)\}$ be a method of C that directly accesses and/or modifies x of C_1. Then, we can refine C and C_1 by changing in class C the method $m()\{c(o.x', o.x)\}$ to $m()\{o.n()\}$ and in class C_1 adding a fresh method $n()\{c[x'/o.x', x/o.x]\}$. This is also called the *expert pattern of responsibility assignment*. This rule and other refinement rules in rCOS can prove big-step refinements, such as the following **expert pattern**, that will be repeatedly used in the design of the system.

Theorem 4 (Expert Pattern) Given a list of class declarations *Classes* and its navigation paths $le \mathrel{\widehat{=}} r_1.\ldots.r_f.x$, $\{a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell\}$, and $\{b_{11}.\ldots.b_{1t_1}.y_1, \ldots, b_{s1}.\ldots.b_{st_s}.y_s\}$ starting from class C, let $m()$ be a method of C specified as

$$\begin{aligned} C :: m()\{ \quad & c(a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell) \\ & \wedge\ le' = e(b_{11}.\ldots.b_{1t_1}.y_1, \ldots, b_{s1}.\ldots.b_{st_s}.y_s)\ \} \end{aligned}$$

Then *Classes* is refined by redefining $m()$ in C and defining the following fresh

methods in the corresponding classes:

$$
\begin{array}{ll}
C :: & check()\{return'{=}c(a_{11}.get_{\pi_{a_{11}}x_1}(), \ldots, a_{\ell 1}.get_{\pi_{a_{\ell 1}}x_\ell}())\} \\
 & m()\{\textbf{if } check() \textbf{ then } r_1.do\text{-}m_{\pi_{r_1}}(b_{11}.get_{\pi_{b_{11}}y_1}(), \ldots, b_{s1}.get_{\pi_{b_{s1}}y_s}())\} \\
T(a_{ij}) :: & get_{\pi_{a_{ij}}x_i}()\{return'{=}a_{ij+1}.get_{\pi_{a_{ij+1}}x_i}()\}\ (i:1..\ell, j:1..k_i-1) \\
T(a_{ik_i}) :: & get_{\pi_{a_{ik_i}}x_i}()\{return'{=}x_i\}\ (i:1..\ell) \\
T(r_i) :: & do\text{-}m_{\pi_{r_i}}(d_{11}, \ldots, d_{s1})\{r_{i+1}.do\text{-}m_{\pi_{r_{i+1}}}(d_{11}, \ldots, d_{s1})\}\ \ (i:1..f-1) \\
T(r_f) :: & do\text{-}m_{\pi_{r_f}}(d_{11}, \ldots, d_{s1})\{x' = e(d_{11}, \ldots, d_{s1})\} \\
T(b_{ij}) :: & get_{\pi_{b_{ij}}y_i}()\{return'{=}b_{ij+1}.get_{\pi_{b_{ij+1}}y_i}()\}\ (i:1..s, j:1..t_i-1) \\
T(b_{it_i}) :: & get_{\pi_{b_{it_i}}y_i}()\{return'{=}y_i\}\ (i:1..s)
\end{array}
$$

where $T(a)$ is the type name of attribute a and π_{v_i} denotes the remainder of the corresponding navigation path v starting at position i. ■

This pattern represents that a computation is realized by obtaining the data that is distributed in different objects via association links and then delegating the computation tasks to the target object whose state is required to change.

If the paths $\{a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell\}$ have a common prefix, say up to a_{1j}, then class C can directly delegate the responsibility of getting the x-attributes and checking the condition to $T(a_{ij})$ via the path $a_{11}.\ldots.a_{1j}$ and then follow the above rule from $T(a_{ij})$. The same optimization can be applied to the b-navigation paths.

The expert pattern is the most often used refinement rule in OO design [32]. It refines a pre-/postcondition functionality specification into object interactions, based on a given class structure. This is an essential difference between the classical refinement in the imperative programming paradigm and in object-oriented programming. The formal definition and proof of it as an OO refinement rule are given in [23,58] using the relational semantics for object-oriented programming.

One feature of this rule is that it does not introduce more couplings by associations between classes into the class structure. It also ensures that functional responsibilities are allocated to the appropriate objects that *know* the data needed for the responsibilities assigned to them. More significantly, we have automated this rule in the rCOS tool [15].

3.2.2 Encapsulation

The *encapsulation rule* says that if an attribute of a class C is only referred to directly in the specification (or code) of methods in C, this attribute can

be made a *private attribute*; and it can be made *protected* if it is only directly referred in specifications of methods of C and its subclasses. We should note that data encapsulation may have to be applied after it is known that new classes introduced in later iterations do not need direct access to or extend the classes of the current iteration.

3.2.3 Class decomposition

During an OO design, we often need to decompose a class into a number of classes. For example, consider classes $C_1 :: D\ a_1$, $C_2 :: D\ a_2$, and $D :: T_1\ x, T_2\ y$. If methods of C_1 only call a method $D :: m()\{...\}$ that only involves x, and methods of C_2 only call a method $D :: n()\{...\}$ that only involves y, we can decompose D into two classes $D_1 :: T_1\ x; m()\{...\}$ and $D_2 :: T_2\ y; n()\{...\}$, and change the type of a_1 in C_1 to D_1 and the type of a_2 in C_2 to D_2. There are other rules for class decomposition in [24].

An important point here is that the expert pattern and the rule of encapsulation can be implemented by automated model transformations. In general, transformations for structure refinement can be aided by transformations in which changes are made with a graphical editing tool, and then automatic transformation can be derived for the change in the specification of the functionality and object interactions. For details, please see our work in [58].

The final model to be produced at the end of the OO design of a use case is a contract in which the use case sequence diagram is refined to an *object sequence diagram* with inter-object interactions, but with the events of interactions between the actors and the system unchanged. The conceptual class diagram is refined to a *design class diagram* in which methods and their specifications are assigned to classes. The correctness of this design model is guaranteed by the correct use of the refinement rules.

3.3 Logical component-based architecture design

In this phase, models of "small use cases" are organised into bigger components by using the union connector, and models of "large use cases" are decomposed into smaller interacting components according to the nature of the objects, such as their physical locations and roles that they play in the overall business organization. As what will be shown in Section 4.4, the interfaces among these components are identified from the object sequence diagrams of the use cases in the design model. The resulting model is called the model of the *logical component-based architecture* and obtained after hiding internal objects and object interactions. The design decisions on (de)composition cannot be completely formalized or automated, but once a decision is made

the construction of the logical component-based architecture from the design model can be automated by model transformations. The model includes a set of components with explicitly specified provided and required interfaces. They can be depicted by a UML *component diagram*, a *component sequence* (or *interaction*) diagram, and a state diagram for each (non-trivial) component. For this model, the *compatibility* of the contracts for the provided and required interfaces of the inter-dependent components should be checked to avoid deadlock and livelock. This is called the logical model because the interfaces of the components are all *object-oriented interfaces* in which component interactions are direct method invocations via object references.

3.4 Detailed design and coding

The detailed design activities, that will be demonstrated later in Section 4.5, include further refinement of the components by class decomposition, data encapsulation and refactoring; and replacement of some of the object-oriented interfaces with concrete and appropriate interaction mechanisms (or middlewares) such as RMI, CORBA, or shared event channels. This transforms the *platform independent model* (PIM) of the logical component-based architecture to a *platform specific model* (PSM) with regard to the interaction mechanisms. Code can be constructed for each component. For this, a code template of each component can be generated from the detailed design of the components. The template includes the information of the flow of method invocations and the assertions specifying the functionality of the methods of the components. The flow of method invocations and assertions is derived and documented during the refinements of the design process. Verification and validation can be applied to components before and after introducing the concrete middlewares, such as runtime checking, testing (unit testing) [34] and even by a *verifying compiler* [26,4].

3.5 Design of GUI and synchronisation with hardware controller

Most software systems nowadays have components of GUI and hardware controllers that interact with the application software components. Some GUI objects and hardware controllers are only relevant to some use cases and the design of the synchronisation among these GUI objects, hardware controllers and the corresponding components can thus be given after the design of these use cases. Others, such as a printer, interact with the components of many use cases and therefore their synchronization with the GUI and components should be designed after the design of all the components. It is important to note that design and analysis of the synchronization among GUI compo-

nents, hardware controllers and logical components can be done in a purely event-based model following the theory of embedded system design. The rCOS method supports the construction and analysis of such a model, but in this paper we will not discuss this with the design of the CoCoME example. A discussion can be found in the full version [14].

4 The Design of CoCoME

The case study of CoCoME is a trading system and extends the running example in Larman's textbook [32], originally called the *Point of Sale* (POS) system. In this section, we apply rCOS with the development process described in the previous section to the design of this system, focusing on one iteration. Subsequent iterations would add more functionality to this core.

4.1 System overview

The trading system is a computerized system typically used in a supermarket. It deals with the various aspects of sales and other business processes, including processing sales at a cash desk and handling both cash and credit card payments, as well as updating the sales at the inventory. Furthermore, the trading system deals with ordering goods from product suppliers (or wholesalers), and generating various kinds of reports for management purposes.

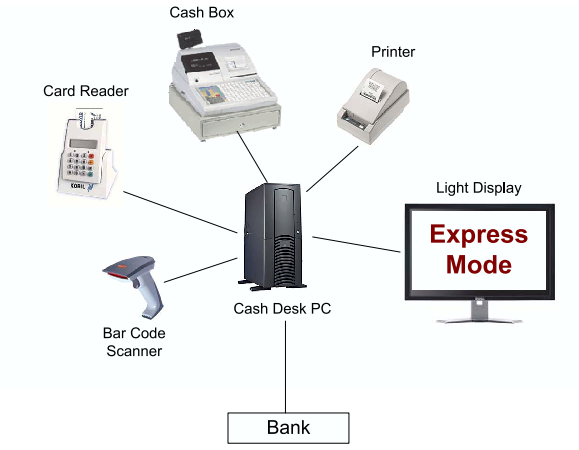


Fig. 7. Overview of the entities of a cash desk (taken from [47])

At the cash desk, the customer can check out the products he wants to buy and make the payment. To do this the cashier records each product by entering the bar code with the bar code scanner or the keyboard. Furthermore, we introduce an express checkout for customers with only a few items in order

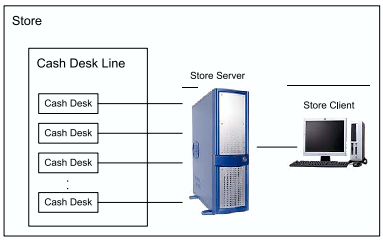


Fig. 8. Overview of a store

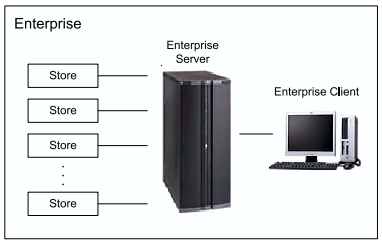


Fig. 9. Overview of an enterprise

to speed up handling customers. A cash desk, shown in Fig. 7, consists of the following devices:

- a *Cash Box* with keys for starting and finishing a sale, entering received money and paying out changes,
- a *Bar Code Scanner* with which product code can be scanned,
- a *Card Reader* for handling credit card payment,
- a *Printer* for printing the receipt at the end of the sale process,
- a *Light Display* which signals whether the cash desk is in express mode,
- a *Cash Desk PC* that integrates the hardware devices and runs the software handling the sale process and communication with the back office and clearing credit card transactions with the *Bank*.

The system can be a large system in which the store has a number of cash desks, called the *cash desk line* for checking out customers in parallel, or even a network of stores, each having a cash desk line, to support a whole enterprise. Each store has its own *Store Server* and *Store Client* for the store management, such as ordering products and managing inventory. Each cash desk of the store is connected to its Store Server (see Fig. 8). In the case of a network of stores, there is need for an *Enterprise Server* and an *Enterprise Client* for enterprise management that we are not concerned with in this paper, and all stores are connected to the *Enterprise Server* (see Fig. 9).

We consider the development of a system that is used in one store, but it has a number of checkout points. It thus includes hardware components such as bar code scanners, card readers, printers, and software to run the system. To handle credit card payments, orders and delivery of products, we assume a *Bank* and a *Supplier* that interact with the system.

4.2 CoCoME requirements modelling and analysis

The system overview and problem description provide the initial context and vocabulary for further requirements elicitation through studying the business processes. The requirements analysis and system design are use-case driven. A *use case* specifies how the system interacts with actors and its environment

in realizing a business process. An actor, such as a *Cashier* who checks out customers, interacts with the system by calling a *system operation* to request a service of the system.

There can be many use cases, depending on what business processes the client wants the system to support. In the following subsections, we work out the model of the use cases that we are concerned with in this paper.

*4.2.1 Use case **UC 1**: Process Sale*

The main use case is about *processing sales*, modelled in use case **UC 1**: *Process Sale*.

Informal description of UC 1 The *normal courses* of interactions between the actors and the system are informally described as follows.

(1) When a *customer* comes to the *cash desk* with her *items*, the *cashier* initiates a new *sale*.
(2) The *cashier* enters each item, either by typing or scanning in the *bar code*; if there is more than one of the same item, the cashier can enter the *quantity*. The system records each item and its quantity and calculates the subtotal.
When the cash desk is operating in *express mode*, only a predefined maximum number of items can be entered.
(3) When there are no more items, the *cashier* indicates to the system *end of entry*. The *total* of the sale is calculated. The cashier tells the customer the total and asks her to pay.
(4) The customer can pay by cash or credit card. If by cash, the amount received is entered. The system records the *cash payment* amount and calculates the change.
If by credit card, the card information is entered. The system sends the credit payment to the *bank* for *authorization*. The payment only succeeds if a positive validation reply is received. In express mode, only cash payment is allowed.
After a successful payment the inventory of the store is updated and the completed sale is logged.

There are *exceptional* or *alternative courses* of interactions, e.g., the entered bar code is not known in the system, the customer does not have enough money for a cash payment, or the authorization reply is negative. A system needs to provide means of handling these exception cases, such as cancel the sale or change to another way of paying for the sale. At the requirements level, we capture these exceptional conditions as preconditions.

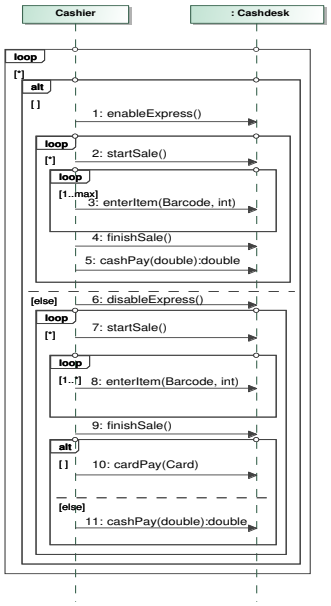


Fig. 10. Sequence Diagram

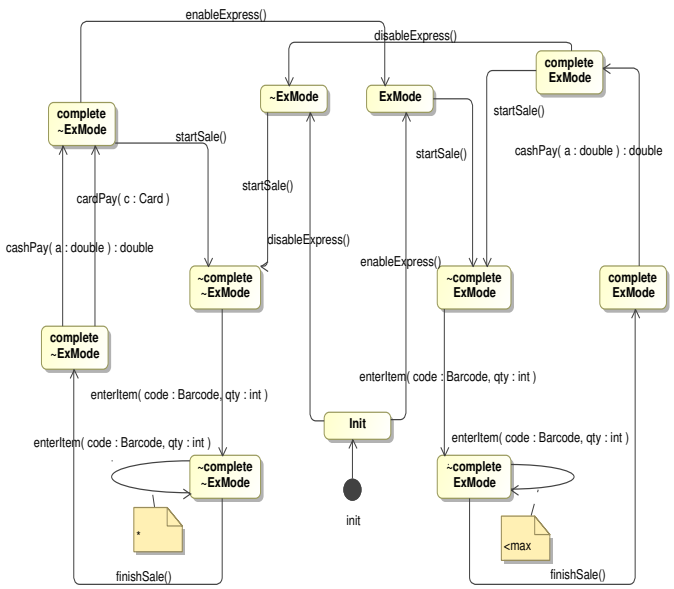


Fig. 11. State Diagram of UC1

Formal model of UC 1 Each use case is modelled by the *contract* of the *provided interface* of a component. Let CashDeskIF denote the provided interface of component *ProcessSale*. We use a schema in the specification of a component and its interfaces in which

- a component can have a number of provided interfaces, whose union defines the overall provided interface of the component,
- similarly a component can have a number (zero or more) of required interfaces whose union defines the overall required interface,
- each interface only declares its methods, and its fields are declared as attributes of a class, called the *interface class*, that *implements* the interface,
- when the contract of the interface is specified,
  - the protocol is a sequence diagram of the interactions between the actors and the instances of the class that implements that interface,
  - the state diagram is the state diagram of instances of this class
  - the functionality of the interface methods are given as the pre and post-conditions of the "definitions" of the methods in the class implementing the interface.

We require that during system execution there is only one instance of the interface class for each component. With this schema and based on the introduction to the system and the use case description, the component and its interface are declared as

```
component ProcessSale {
  provided interface CashDeskIF {
    public enableExpress();
    public disableExpress();
    public startSale ();
    public enterItem(Barcode code, int qty);
    public finishSale ();
    public cardPay(Card c);
    public cashPay(double a ; double c);
  }
}
```

The protocol of CashDeskIF is modelled by the sequence diagram in Fig. 10 and the dynamic flow of control by the state diagram in Fig. 11. The invariant and static functionality of the methods of the interface are specified as follows.

class *CashDesk* **implements** *CashDeskIF*::
invariant $store \neq null \wedge store.catalog \neq null$
$\wedge\ (exmode = true \vee exmode = false)$
method *enableExpress*()
pre: $true$
post: $exmode := true$
method *disableExpress*()
pre: $true$
post: $exmode := false$
method *startSale*()
pre: $true$
post: /* a new *sale* is created, and its *line items* initialized,
and the date correctly recorded */
$sale := Sale.New(false/complete, empty/lines, clock.getDate()/date)$
method *enterItem*(*Barcode code, int qty*)
pre: /* there exists a product with the input barcode *c* */
$store.catalog$.find(code) $\neq null$
post: /* a new *line* is created with *barcode c* and *quantity qty* */
$LineItem\ line := LineItem.New(code/barcode, qty/quantity)$
; $line.subtotal := store.catalog$.find($code$).$price \times qty$
; $sale.lines$.add($line$)
method *finishSale*()
pre: $true$
post: $sale.complete := true$
$\wedge\ sale.total := sum[[l.subtotal \mid l \in sale.lines]]$
method *cashPay*(*double a; double c*)
pre: $a \geq sale.total$
post: $sale.pay := CashPayment.New(a/amount, a\text{-}sale.total/change)$
$\wedge\ c := a - sale.total$
/* the completed *sale* is logged in *store*, and */
; $store.sales$.add($sale$) /* the inventory is updated */
; $\forall l \in sale.lines,\ p \in store.catalog \cdot$ (**if** $p.barcode = l.barcode$ **then**
$p.amount := p.amount - l.quantity$)

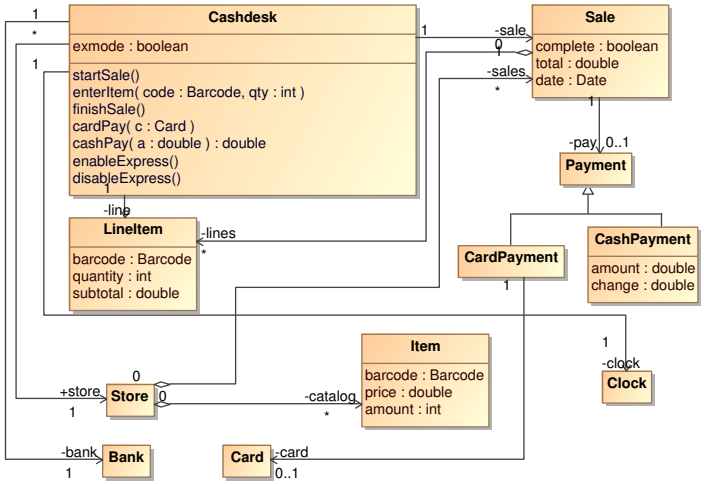


Fig. 12. Class Diagram of UC1

method $cardPay(Card\ c)$
pre: $Bank.authorize(c, sale.total)$
post: $sale.pay := CardPayment.New(c/card)$
; $store.sales.\text{add}(sale)$
; $\forall l \in sale.lines,\ p \in store.catalog \cdot (\textbf{if}\ p.barcode = l.barcode\ \textbf{then}$
$p.amount := p.amount - l.quantity)$

Locally declared variables, such as *line*, are not part of the interface. Also notice that we have used $le := e$ according to its semantics in UTP to avoid the use of a frame in the corresponding predicate $\beta : le' = e$.

The datatypes and classes of the objects are declared as class declarations in rCOS and are represented by the UML class diagram in Fig. 12. Then the *state space* of the component is formalized in rCOS [23], and each state can be represented as a UML object diagram.

Notice that the exception conditions are specified in the preconditions of the methods, and an invocation with input that falsifies the precondition will lead the execution to divergence. What to do when an exception occurs can be part of the client's requirements. For example, when the bank authorisation for a card payment is refused the actor can decide to *cancelSale*() or use *cashPay*(). If this is specified in the sequence diagram and state diagram, the functionality of *cancelSale*() should also be specified, that is **skip** in this case, as the inventory update is done only by the payment methods.

Analysis and checking Static consistency between methods in the diagrams and the functional specification, their types, and navigation paths must be checked. This step is usually done by tools like a compiler.

Dynamic consistency ensures that the separately specified behavior in the sequence diagram and the state diagram are consistent. Informally, the consistency must ensure that whenever the actors follow the interaction protocol defined by the sequence diagram, the interactions will not be blocked by the system, i.e. no deadlock should occur. Formally speaking, this requires that the traces of the sequence diagram are accepted by the state machine defined by the state diagram.

In the CoCoME exercise, we translated both the sequence diagram and the state diagram into CSP processes [25] and used the CSP model checker FDR [48] to check that the CSP process for the state diagram is trace equivalent to the CSP process of the sequence diagram. In the case when the CSP process of the sequence diagram defines a regular language, as is the case for use case **UC 1**, we define the semantics of a use case sequence diagram by the set of traces of the diagram expressed by the regular expression:

$$
\begin{aligned}
&tr(SD_{uc1}) = \\
&(\quad enableExpress()\ \ (startSale()\ enterItem()^{(max)}\ finishSale()\ cashPay())^* \\
&\ \ + \ disableExpress()\ \ (startSale()\ enterItem()^{+}\ finishSale()\ (cashPay()\ +\ cardPay()))^*\ )^*
\end{aligned}
$$

For this case, we can also check whether the state machine defined by the state diagram accepts this regular language. Notice that we have a single sequence diagram for each use case, that represents the protocol in which the actor interacts with the component. We do not define the composition of sequence diagrams at the level of the UML profile, but we have compositions of the translated protocols defined in rCOS.

While the sequence diagram specifies the traces in a denotational manner, the state diagram describes the operational flow of control and thus model checking and simulation can be applied. The state diagram allows verification of both safety and liveness properties. For example, we can check with a model checking tool that the state diagram satisfies the properties that

(1) in the express mode, i.e. when $exmode = true$, at most max items can be entered, and if an event of *enterItem*() occurs, it will eventually be followed by a *cashPay*() event;
(2) after an *enterItem*() event, an event of *cashPay*() or *cardPay*() will occur;
(3) in a sale, a *cashPay*() or a *cardPay*() cannot occur before *finishSale*().

Notice that properties (1) and (2) are only an abstraction of the property that *every item sold will be paid when the sale completes*. With the event based model of the protocol and the state machine these properties do not ensure that *every item sold will be paid with the correct amount when the sale completes*. To prove that each item will be paid with the correct amount, we need to analyse the functional specifications of the methods.

Validation of the functionality specifications is a difficult issue, its completeness in particular. We apply a prototyping tool [35] to validate the model of a use case. Formal deductive proof techniques aided by a theorem prover can be used to prove some functional properties. For example, from the specification of *enterItem*(), *finishSale*() and *cashPay*() and *cardPay*(), we can prove that every item entered is recorded in the current sale with the correct subtotal, the sale's total is correctly calculated, and the sale's total is correctly paid for by the payment. This, together with properties (1) and (2) proved for the dynamic behavior, proves the property that *each item sold is paid with the correct amount when the sale completes.*

4.2.2 Two more use cases

To expose the most significant design issues and techniques, we consider two more use cases of the CoCoME exercise. These two use cases are simple and there is no need to give their sequence-, state- and class diagrams. Further, we will directly write the functionality of the methods of the use cases together with their corresponding component and interface declarations.

UC 2: Order product This use case starts with *startOrder* and then a number of times *orderItem* followed by *makeOrder*. We only specify the functionality of the use case operations, and omit the declaration of the related classes which are obvious. We use the name *OrderProduct* for the component of this use case and let a supplier provide the service *receiveOrder*(*Order order*) to the store:

```
component OrderProduct {
  provided interface OrderDeskIF {
    public startOrder();
    public orderItem(Barcode c, int q);
    public makeOrder(Order order);
  }
  class OrderDesk implements OrderDeskIF {
    protected Store store;
    protected Order order;
    public startOrder() {
                  pre: true
                  post: /* a new order is created */
                        order := Order.New()
                   }
    public orderItem(Barcode c, int q) {
                  pre: /* the product code c exists in the catalog */
                        store.supplier.catalog.find(c) != null
                  post: /* create an order line and add it to the order */
                         Orderline line := OrderLine.New(c/identifier, q/quantity)
                         ; order := order.lines.add(line)
                  }
    public makeOrder(Order order) {
                  pre: true
                  post: store.order.add(order)
                         ∧ store.supplier.receiveOrder(order)
                  }
 }
}
```

UC 3: Manage inventory This use case carries out changes to the inventory items. Here we only specify the operations for changing the price of an item and adding a new item. Also, the protocol of this use case allows any sequence of invocations of these operations and is thus omitted.

```
component InventoryManagement {
  provided interface InventoryDeskIF {
    public changePrice(Barcode code, double newPrice);
    public addItem(Barcode c, int a, double p);
  }
  class InventoryDesk implements InventoryDeskIF {
    protected Store store;
    public changePrice(Barcode code, double newPrice) {
                  pre:  store.catalog.find(code) != null
                  post: ∀ p ∈ catalog · (if p.barcode=code then p.price := newPrice)
                }
    public addItem(Barcode c, int a, double p) {
                  pre:  valid(c) /* the product code c is valid */
                  post: store.catalog.add(Item.New(c/barcode, a/amount, p/price))
                }
 }
}
```

In the trading system, there are many other methods in the store management component, including changing the amount of an item, changing the price of a product, and deleting a product, not to say producing many different reports. However, the selection above is sufficient to illustrate the rCOS approach.

4.2.3 Refinement of use case

Further analysis of use case **UC 1** allows us to introduce two sub-use cases, **UC 1.1**: *Make Cash Payment* and **UC 1.2**: *Make Card Payment*. Use case **UC 1.1.** provides the master use case **UC 1** an interface CashPaymentIF with a method *cashPay*(*Sale sale,* **double** *a;* **double** *c*) which is specified in the same way as the method *cashPay*(**double** *a;* **double** *c*) in the contract for CashDeskIF. In the same way, **UC 1.2** provides the master use case **UC 1** with interface CardPaymentIF containing a method *cardPay*(*Sale sale, Card c*), specified in the same way as the method *cardPay*(*Card c*) in the contract for CashDeskIF. We then obtain a refined model of use case **UC 1**:

```
component ProcessSale {
 /* wrap class in component: */
 component CashDeskComp {
   required interface CashPaymentIF;
   required interface CardPaymentIF;
   provided interface CashDeskIF {
     public enableExpress();
     public disableExpress();
     public startSale();
     public enterItem(Barcode code, int qty);
     public finishSale();
     public cardPay(Card c);
     public cashPay(double a; double c);
   }
   class CashDesk implements CashDeskIF {
     protected boolean exmode;
     protected Store store;
     protected Sale sale;
   } } /* CashDeskComp */
 component CashPayment {
  provided interface CashPaymentIF {
    public cashPay(Sale sale, double a; double c)
 } }
 component CardPayment {
  provided interface CardPaymentIF
     { public cardPay(Sale sale, Card c) }
} } /* ProcessSale */
```

This composite component can be designed in rCOS by the parallel union and hiding the shared interface operations as:

$$ProcessSale \mathrel{\widehat{=}} (CashDeskComp \parallel CashPayment \parallel CardPayment) \setminus (\textsc{CardPaymentIF} \cup \textsc{CashPaymentIF})$$

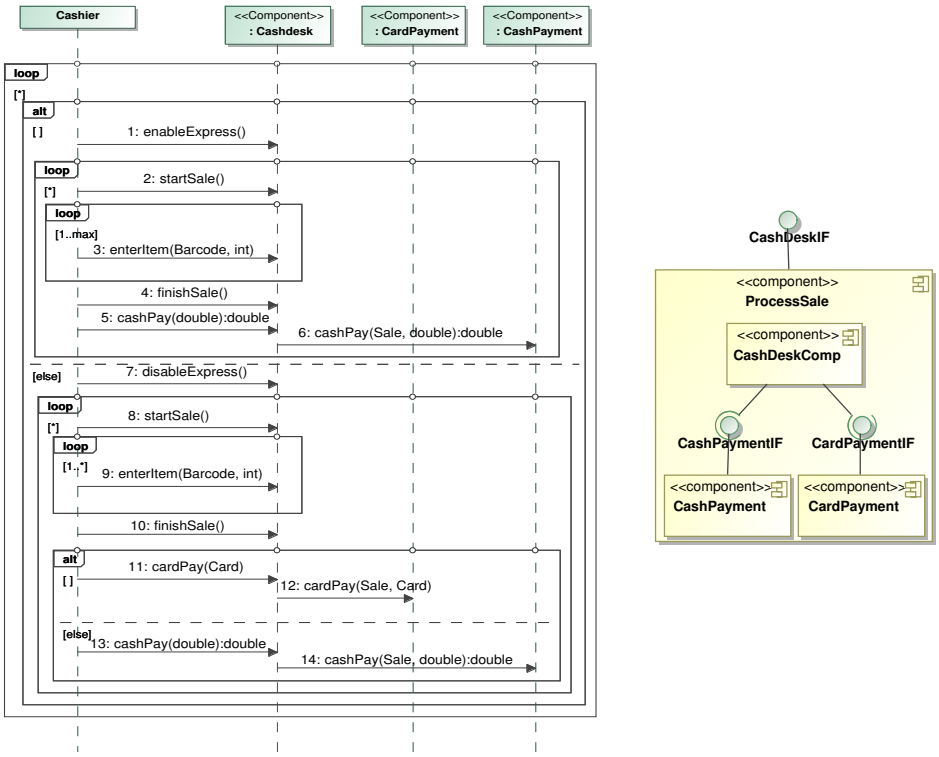


Fig. 13. Refined sequence and component diagram of **UC 1**

The original use case sequence diagram in Fig. 10 is then refined to the use case sequence- and component diagram in Fig. 13. As the two sub-use cases are very simple, their state diagrams do not play an important role in further design. Therefore, we do not need to change the state diagram of **UC 1**. A slight change to the class diagram is needed so that the interface classes of the sub-use cases can be included. Of course a different refinement can be applied; this depends on the designer's decision.

4.2.4 Integrating the models and global constraints

We have captured three use cases and each is modelled as a contract of the provided interface of a component. At this level, they look like separate closed components. In fact, they are not independent but share data and objects. We need to analyze their relation and identify the objects that they share. We can see, from the attributes of the classes implementing the interfaces that all the three components use an object *store* with the type *Store*. Obviously,

only when the *store* objects of the three components are the same, they can be composed into a component that supports the application:

$$StoreComp \mathrel{\widehat{=}} ProcessSale \parallel OrderProduct \parallel InventoryManagement$$

In a component with name C and some interface IF, let $C.IF.a$ denote a *public* or *protected* attribute $T\ a$ of the (abstract) class implementing the interface IF. We omit IF or C where there is only one such $T\ a$ in all implementing classes. We can then add a constraint on the relation among the subcomponents of the composite component *StoreComp* specified as an invariant

component *StoreComp* { **invariant**
$null \neq ProcessSale.store{=}OrderProduct.store{=}InventoryManagement.store$ }

The constraints have to be established when the system is integrated and the components, i.e. the instances of the interface classes, are created. They must be preserved by the execution of the methods of the interface objects.

These constraints have to be guaranteed by the functional behavior of the components and by the consistency of the class models of the components. For example, the constraint on *StoreComp* requires that the *Store* class is defined to be the same in all the three subcomponents. This leads to a problem of how the class diagram of the system evolves during incremental requirements modelling and analysis. Our approach is to use one global set of classes for all the use cases. The set evolves by adding new classes, new attributes and new associations, when a new use case is specified. In this way, we can easily ensure the consistency of the class definitions as all these kinds of evolution of the global set are refinements to the class structure [23,58]. For example, the new classes, *Order* and *Orderline* and their associations with the *Store* and *Product*, used in the specification of the methods of use cases **UC 2** and **UC 3** must be added to the global set containing the elements in Fig. 12.

4.2.5 Discussion about requirements modelling and analysis

With an informal description, we are not able to describe the functionality of a use case completely and clearly. Yet, it would be too complex to describe in a single notation the interactions, dynamic flow of control and synchronization, functionality and the static class structure. rCOS gives a clear separation of these views. Obviously, changes in the interaction protocol and the dynamic flow of control are required to be consistent, but they do not lead to changes in the static functionality of the methods and the class structure. Also, the change of functionality specification of the methods and the class diagram should be made consistently, but these do not affect the change of the protocol and the dynamic flow of control. The consistency between the interaction protocol

and the state diagram is mainly semantic, whereas the consistency between the functionality specification and the class diagram is mainly syntactic.

A thorough and precise domain investigation is crucial to establish the *vocabulary* of the application. This vocabulary is used to define the interface methods, classes and objects of the use cases. A precise model of the interactions of a use case is the first key step to identify and define the interface methods. The precise specification of the functionality of the interface methods is important for the construction of the class diagram that defines the classes and their attributes and associations. The specification of the functionality also determines later in the design how objects should interact with each other to realize the specified use cases. Therefore, without a full specification of the use case interaction protocol, the functionality, the classes and their attributes and associations, it would be difficult to enter the design phase.

4.3 Design of the functionality of CoCoME by refinement

This section illustrates how refinement rules in rCOS, the rule for the **expert pattern** in particular, are effectively used to work out a correct design for CoCoME. We focus on the design methods of use case **UC 1**. We use the convention $ClassC :: m()\{c\}$ of Java to denote that method $m()$ of class C is defined by the command c (it can also be a specification statement).

Operations *enableExpress*() and *disableExpress*() The postconditions of these two methods of class *CashDesk* modify the attribute *display* of object *light*. Through the expert pattern, we introduce two methods into class *Light*, and then refine the specification of *enableExpress*() and *disableExpress*() accordingly:

Class *CashDesk*:: *Light light* /* attribute */
enableExpress() {*exmode* := *true*; *light.turnGreen*()}
disableExpress() {*exmode* := *false*; *light.turnYellow*()}
Class *Light*:: *Color display* /* attribute */
turnGreen(){*display* := *green*}
turnYellow(){*display* := *yellow*}

This refinement also leads to refinement of the sequence diagram in Fig. 10 extending it with an interaction between *:CashDesk* and *:Light* representing an invocation of *turnGreen*() of *:Light* by the method *enableExpress*() of *:CashDesk*, and an interaction representing an invocation of *turnYellow*() of *:Light* by the method *disableExpress*() of *:CashDesk*. We are implementing an algorithm to automate the refinement by the expert pattern. We plan to develop a *design*

by drawing facility in the tool in order for these steps of refinement to be done by drawing on the sequence diagram.

For these refinement steps of the methods, the class diagram in Fig. 12 is also refined by adding the definitions of *turnGreen*() and *turnYellow*(), as well as replacing the bodies of methods *enableExpress*() and *disableExpress*() with the refined definitions in class *CashDesk*. This means that the refinement of the functionality of the methods and the change of the class diagram have to be carried out hand in hand consistently. The UML profile and the rCOS tool ensure that the change in the sequence diagram will be automatically applied to the class diagram, too. In the refinement of the other methods of use case **UC 1**, we assume the same understanding of how the refinement derives consistent changes in the sequence diagram and class diagram.

Operation *startSale*() The specification says to get the date by *clock.getDate()* and create a new *sale*. The expert pattern allows to delegate responsibility for getting the date to the clock and responsibility for creating the new *sale* to the class *Sale*:

$$\begin{array}{ll} \textbf{Class } \mathit{CashDesk}:: & \mathit{startSale}()\ \{\mathit{sale} := \mathit{Sale.New}(\mathit{clock.getDate}())\} \\ \textbf{Class } \mathit{Clock}:: & \mathit{getDate}(\ ;\ \mathit{Date\ return})\ \{\mathit{return} := \mathit{date}\} \end{array}$$

The above definition of *startSale*() can be further refined to

$$\textbf{Class } \mathit{CashDesk}::\mathit{startSale}()\ \{\mathit{Date\ d} := \mathit{clock.getDate}(); \mathit{sale} := \mathit{Sale.New}(d)\}$$

This specification uses the constructor method of class *Sale* which also takes an input parameter *Date d*. In general, a constructor method of a class defines the initial values of the attributes of the class. For class *Sale*, the attributes are *complete*, *lines*, *date*, *pay* and *total*. Notice that the type of attribute *lines* is a set of *LineItem*. In Java, sets are implemented by classes that implement the interface SET. The constructor of the set class initializes the instance as an empty set. For a set class $set(T)$, we use the Java notation of $s := set(T).New()$ and its semantics (definable in rCOS as $true \vdash s' = \emptyset$) for the creation of a set object. An object is always created by the constructor of the class (this is a special case of the expert pattern). According to the specification, the constructor *Sale*() is thus defined as:

$$\begin{array}{ll} \textbf{Class } \mathit{Sale}:: & \mathit{Sale}(\mathit{Date\ d})\ \{\ \ \mathit{lines} := \mathit{set}(\mathit{LineItem}).\mathit{New}(); \\ & \qquad\qquad \mathit{total} := 0; \mathit{date} := d; \mathit{complete} := \mathit{false}\ \} \end{array}$$

We obtain the following design after the above refinement steps:

$$\begin{array}{ll} \textbf{Class } \mathit{CashDesk}:: & \mathit{Clock\ clock}; \\ & \mathit{startSale}()\ \{\ \mathit{Date\ d} := \mathit{clock.getDate}(); \mathit{sale} := \mathit{Sale.New}(d)\} \\ \textbf{Class } \mathit{Sale}:: & \mathit{Sale}(\mathit{Date\ d})\{\mathit{lines} := \mathit{set}(\mathit{LineItem}).\mathit{New}(); \\ & \qquad\qquad \mathit{total} := 0;\ \mathit{date} := d;\ \mathit{complete} := \mathit{false}\} \\ \textbf{Class } \mathit{Clock}:: & \mathit{date}(\ ;\ \mathit{Date\ return})\{\mathit{return} := \mathit{date}\} \end{array}$$

Operation *enterItem*() This is the first time we meet a non-trivial precondition in a specification: $store.catalog.find(c) \neq null$. We introduce a refinement schema, denoted by **PP**.

$$(\mathbf{PP}):\ m()\{\mathbf{pre}: p; \mathbf{post}: R\} \sqsubseteq m()\{\mathbf{if}\ p\ \mathbf{then}\ R\ \mathbf{else\ throw\ exception}(p)\}$$

where **throw exception**(p) can be any specification of what to do when the precondition does not hold, and the worst case is **chaos**. After this refinement, we can apply the expert pattern separately to the refinement of the expression for the precondition and for the postcondition respectively.

Applying the expert pattern to the precondition of *enterItem*(), we need to define the following methods:

Class *CashDesk*:: *find*(*Barcode code* ; *Item return*) {*return* := *store.find*(*code*)}
Class *Store*:: *set*(*Item*) *catalog*;
find(*Barcode code* ; *Item return*) {*return* := *catalog.find*(*code*)}
Class *set*(*Item*):: *find*(*Barcode code* ; *Item return*) {*return* := *find*(*code*)}

Here, we assume the method *find*(*code*) of *set*(*Item*) implements the specification that returns the item with the bar code equal to *code*. The implementation requires the design of an algorithm for the specification and thus involves *significant algorithm design*. The code of the *find*() method in *CashDesk* and *Store* can be automatically generated from the expert pattern, but the code for *find*() of *set*(*Item*) has to be programmed and verified against its specification.

The method signature *makeLine*(*Barcode c,* **int** *qty*) denotes the method realising the postcondition of *enterItem*() when its precondition holds. We apply the expert pattern to refine each conjunct in the specification of the postcondition using proper sequential composition:

Class *CashDesk*:: *makeLine*(*Barcode c,* **int** *qty*) {
line := *LineItem.New*(*c*, *qty*);
line.subtotal(*store.getPrice*(*c*),*qty*); *sale.addLine*(*line*) }
Class *Store*:: *set*(*Item*) *catalog*;
getPrice(*Barcode c* ; **double** *return*)
{*return* := *catalog.getPrice*(*c*)}
Class *set*(*Item*):: *getPrice*(*Barcode c* ; **double** *return*)
{*return* := *find*(*code*).*price*}
Class *Sale*:: *set*(*LineItem*) *lines*;
addLine(*LineItem l*){*lines.add*(*l*)}
Class *LineItem*:: *LineItem*(*Barcode c*, **int** *qty*){*barcode* := *c*; *quantity* := *qty*};
subtotal(**double** *price*, **int** *qty*){*subtotal* := *qty* × *price* }

Here again *lines.add*(l) implements the specification $lines' = lines \cup \{l\}$. We leave the design for exception handling unspecified. Any decision is formally a refinement

of the original specification. Further refactoring [19] can introduce more methods to a class so that method calls do not occur in method parameters and this will be discussed in Subsection 4.5. Correct refactoring is also formalized as refinement in rCOS [23].

With the definition of the methods for the pre and postcondition of *enterItem*(), we apply the refinement rule (**PP**) and obtain the following design:

$$\begin{aligned}&\textbf{Class } \mathit{CashDesk}:: \mathit{enterItem}(\mathit{Barcode}\ \mathit{code}, \textbf{int}\ \mathit{qty})\{\\ &\qquad\qquad \textbf{if } \mathit{find}(\mathit{code}) \neq \mathit{null} \textbf{ then } \mathit{makeLine}(\mathit{code}, \mathit{qty})\\ &\qquad\qquad \textbf{else throw exception } (\mathit{find}(\mathit{code}) \neq \ \mathit{null})\}\end{aligned}$$

Operation *finishSale*() We now refine method *finishSale*() using the expert pattern and define a method *setComplete*() and a method *setTotal*() in class *Sale*. These methods will be called by the use case handler class. We omit discussion of *addAll*(), as we will discuss quantification over sets separately in the following.

Class *CashDesk*::	*finishSale*()	{ *sale.setComplete*(); *sale.setTotal*() }
Class *Sale*::	*setComplete*()	{ *complete* := *true* }
	setTotal()	{ total := *lines.addAll*() }
Class *set*(*LineItem*)::	*addAll*(; **double** *return*)	{ /* elided */ }

Operation *cashPay*() According to the refinement rule (**PP**) and the expert pattern, for the precondition $a \geq \mathit{sale.total}$, we introduce the method *checkPre*() into *CashDesk* and *getTotal*() into *Sale*:

$$\begin{aligned}&\textbf{Class } \mathit{CashDesk}:: \mathit{checkPre}(\textbf{double}\ a\ ; \textbf{boolean}\ \mathit{return})\\ &\qquad\qquad \{\mathit{return} := a \geq \mathit{sale.getTotal}()\}\\ &\textbf{Class } \mathit{Sale}:: \quad \mathit{getTotal}(\ ;\ \textbf{double}\ \mathit{return})\{\mathit{return} := \mathit{total}\}\end{aligned}$$

The first conjunct of the postcondition can be designed easily in the same way as the methods designed earlier. Consider the last part in the postcondition that updates the inventory. It involves quantifications over elements of sets:

$$\begin{aligned}&\forall l \in \mathit{sale.lines}, p \in \mathit{store.catalog} \cdot (\textbf{if } p.\mathit{barcode} = l.\mathit{barcode}\\ &\textbf{then } p.\mathit{amount} := p.\mathit{amount} - l.\mathit{quantity})\end{aligned}$$

In general, for a specification of the form $\forall\ T\ o \in s \cdot \mathit{statement}(o)$ for s being a set of type $\mathit{set}(T)$, we introduce the following refinement rule, called the *universal quantification pattern* (**UQP**):

$$\begin{aligned}&(\textbf{UQP}) : \forall\ T\ o \in s \cdot \mathit{statement}(o) \sqsubseteq\\ &\qquad\qquad \mathit{Iterator}\ i := s.\mathit{iterator}();\\ &\qquad\qquad \textbf{while } i.\mathit{hasNext}()\ \{T\ o := i.\mathit{next}(); \mathit{statement}(o)\}\end{aligned}$$

This means that we should define the semantics of the "Java" statements on the right hand side of the above refinement in rCOS as $\forall\ T\ o \in s \cdot statement(o)$, where $statement(o)$ is an rCOS statement. Now following the expert pattern and (**UQP**), we define the update methods in these two classes:

```
Class CashDesk:: updateInventory() {
                   set(LineItem) lines := sale.getLines();
                   Iterator i := lines.iterator();
                   while i.hasNext() {LineItem l := i.next();
                     store.updateInventory(l.code, l.quantity) }
                 }
Class Store::    updateInventory(Barcode c, int qty) {
                   Iterator i := catalog.iterator();
                   while i.hasNext() {Product p := i.next();
                     if p.code = c then p.amount := p.amount − qty}
                 }
```

The use of (**UQP**) shows the advantage of the combination of rCOS refinement rules and advanced features and libraries implemented in modern languages, such as Java. Obviously, the application of (**UQP**) with the expert pattern is not trivial, one can easily come up with the design in the following example:

Example 2 (A Flawed Design)

```
Class CashDesk:: updateInventory() {sale.updateInventory() }
Class Sale::     set(LineItem) lines;
                 updateInventory() {Iterator i := lines.iterator();
                   while i.hasNext(){LineItem l := i.next();
                     store.updateInventory(l.code, l.quantity)
                 }
Class Store::    updateInventory(Barcode c, int qty) {
                   Iterator i := catalog.iterator();
                   while i.hasNext() { Product p := i.next();
                     if p.code = c then p.amount := p.amount − qty }
                 }
```

The problem with this design is that in the class diagram there is no association from class Sale to class Store, and thus the sale object cannot call a method of the store object. Of course, we can add such an association to make this design work, but this would make the coupling higher, and every time a sale would be created by the cashdesk object, the store object should be passed as a parameter to the constructor of Sale. ■

The example shows that the expert pattern must be applied in the context of the class diagram. It is important to note that the "flawed design" can be syntactically

detected when checking the consistency between the functionality specification and the class diagram. However, we suggest that checking is done before and during the refinement. Such mistakes can also be avoided in an integrated tool.

How to refine a specification of the form $\exists\ To \in s \cdot statement(o)$? In general, this can be directly specified as a *non-deterministic* statement $\sqcap_{o \in s} statement(o)$ which can then be refined to reduce non-determinism. This in general requires the design of the data structure and algorithm. For example, with an array one can select the first element such that $statement(o)$ holds. However, if p is a property on the elements of s such that there is only one o in s such that $p(o)$ holds and the statement is of the form **if** $p(o)$ **then** $statement(o)$, we can have the following design pattern

$$
\begin{aligned}
(\mathbf{EQP}) : \exists\ T\ o \in s \cdot\ & \mathbf{if}\ p(o)\ \mathbf{then}\ statement(o) \sqsubseteq \\
& \mathbf{boolean}\ b := true; Iterator\ i := s.iterator(); \\
& \mathbf{while}\ \ i.hasNext() \wedge b \\
& \quad \{T\ o := i.next(); \mathbf{if}\ p(o)\ \mathbf{then}\ \{b := false; statement(o)\}\}
\end{aligned}
$$

The pattern must be used with the verification of the invariant that one and only one element of s satisfies property p.

Summarising the design The design of operation *cardPay*() and the other two use cases can be carried out in a similar way, as we have used for the operations that we have just designed. The design of the use cases derives refined models of the use case sequence diagrams, called *design sequence diagrams*. The refined sequence diagram of use case **UC 1** is given in Fig. 14. The definitions of the methods of the classes in the design process can also be collected into the class diagram in Fig. 12, transforming the *conceptual class diagram* in the requirements model to a *design class diagram* shown in Fig. 15.

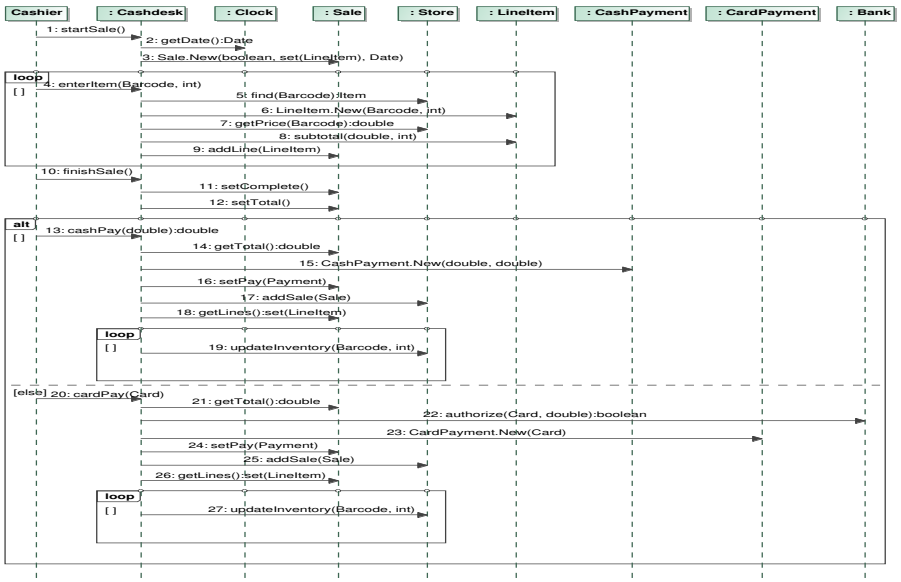
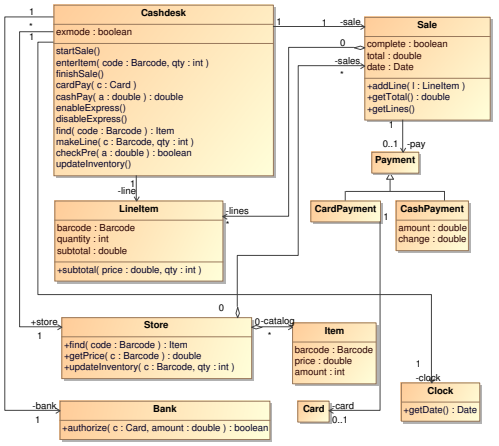


Fig. 14. Design Sequence Diagram



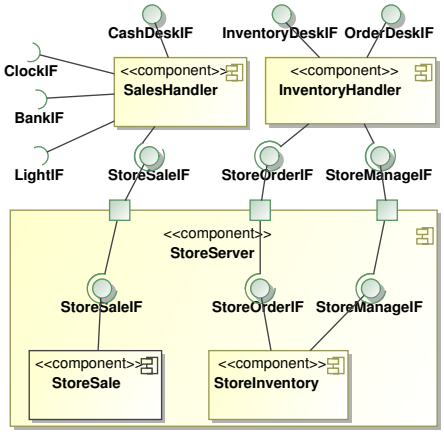


Fig. 15. Design Class Diagram of **UC 1**

Fig. 16. Logical component model

4.4 Logical component-based architecture of CoCoME

After the object-oriented design of the use case methods, we design the component-based architecture. The components we have seen until now are directly derived from use cases: each (sub-)use case defines one component.

We now *decompose* some use case components into a number of components plugged together, and *union* a number of simple components into a single, larger component.

In an analysis of the model of **UC 1**, we can identify objects *clock*, *light* and *bank* that are unique and permanent in the component. Their uniqueness and permanency can be verified as an invariant property, and they can thus be treated as components. Furthermore, they should be treated as components because they are either given as external components (the bank) with known interfaces, or a physical component for which a controller exists has to be designed (the light) or a component with well-known implementation (the clock). We therefore transform these objects into components with their interfaces, denoted by ClockIF, LightIF and BankIF. They consist of the methods being identified as those in the design sequence diagram that are invoked by the other objects of the use case component. In fact the corresponding classes *Clock*, *Light*, and *Bank* from the original model can be used as the classes implementing these interfaces.

There is another unique and permanent object, that is the *store*, which provides much functionality to the other objects. Furthermore, there is also a global constraint on the three use cases that they all share a single *store* object. We therefore make this object into a component called *StoreSale*, together with the objects *catalog* and *sales* that the store aggregates. Their respective aggregated objects *line items* and *products* and *payments* are assigned to this component as well. The interface of *StoreSale*, denoted by StoreSaleIF, consists of those methods of use case **UC 1** in the design sequence diagram Fig. 14 that the component *CashDesk* invokes on the objects now bundled in *StoreSale*. Method invocations that are now component-internal, like those from *store* to the *products*, are not visible.

After identifying the above four components, the temporary objects *line*, *sale*, and *pay* form an open component, denoted as *SalesHandler*, which has the provided interface CashDeskIF and required interfaces ClockIF, LightIF, BankIF and StoreSaleIF, and we have

$$\begin{aligned} \mathit{ProcessSale} \mathrel{\widehat{=}} \; & \mathit{SalesHandler} \parallel \mathit{StoreSale} \parallel \mathit{Clock} \parallel \mathit{Light} \\ & \setminus (\textsc{SalesHandlerIF} \cup \textsc{StoreSaleIF} \cup \textsc{ClockIF} \cup \textsc{LightIF}) \end{aligned}$$

This component is still open and has the required interface BankIF since we assume the existence of the bank server.

Because of the nature and physical location of the use cases *OrderProduct* and *InventoryManagement*, we combine them into a single component:

$$\mathit{Inventory} \mathrel{\widehat{=}} \mathit{OrderProduct} \parallel \mathit{InventoryManagement}$$

This component can then be specified as

```
component Inventory {
  provided interface OrderDeskIF
  provided interface InventoryDeskIF;
  required interface StoreSaleIF;
  class OrderDesk implements OrderDeskIF {
  /* copied from component OrderProduct */
    protected Store store;
    protected Order order;
    public startOrder() {
      pre: true
      post: /* a new order is created */
            order := Order.New()
    }
    public orderItem(Barcode c, double q) {
      pre: /* the product code c exists in the catalog */
           store.supplier.catalog.find(c) != null
      post: /* create an order line and add it to the order */
            OrderLine line := OrderLine.New(c/identifier,q/quantity)
            ; order := order.lines.add(line)
    }
    public makeOrder(Order order) {
      pre: true
      post: store.order.add(order)
            ∧ store.supplier.receiveOrder(order)
    }
  } /* OrderDesk */
  class InventoryDesk implements InventoryDeskIF {
    /* copied from component InventoryManagement */
    protected Store store;
    public changePrice(Barcode code, double newPrice) {
      pre:  store.catalog.find(code) != null
      post: ∀ p ∈ store.catalog · (if p.barcode=code then p.price:=newPrice)
    }
    public addItem(Barcode c, long a, double p) {
      pre:  valid(c) /* the product code c is valid */
      post: store.catalog.add(Item.New(c/barcode, a/amount, p/price))
    }
   } /* Inventory desk */
  invariant ∃ Store s · s = OrderDesk.store = InventoryDesk.store;
}
```

If one wants to hide the individual interfaces by merging them into one, the union

connector and proxy methods can be specified in the following format that also indicates a Java implementation.

```
component Inventory {
  required interface OrderDeskIF;
  required interface InventoryDeskIF;
  invariant ∃ Store s · s = OrderDeskIF.store = InventoryDeskIF.store;
  provided interface InventoryIF {
    public changePrice(Barcode code, double newPrice);
    public addItem(Barcode c, long a, double p);
    public startOrder();
    public orderItem(Barcode c, double q);
    public makeOrder(Order order);
  }
  public class InventoryHandler implements InventoryIF {
    protected InventoryDesk inventoryDsk = InventoryDesk.New();
    protected OrderDesk orderDsk = OrderDesk.New();
    public changePrice(Barcode code, double newPrice)
              { inventoryDsk.changePrice(long code, double newPrice) }
    public addItem(Barcode c,long a, double p)
              { inventoryDsk.addItem(long c, long a, double p) }
    public startOrder() { orderDsk.startOrder() }
    public orderItem(Barcode c, double q) { orderDsk.orderItem(long c, double q) }
    public makeOrder(Order order) { orderDsk.makeOrder(Order order) }
  }
 }
```

In the same consideration as to the decomposition of component *ProcessSale* into *SalesHandler* and *StoreSale*, we decompose *Inventory* into *InventoryHandler* and *StoreInventory*. *InventoryHandler* provides the two interfaces OrderDeskIF and InventoryDeskIF of *Inventory*, and consists of the interface objects of these interfaces. It requires the interfaces denoted by StoreOrderIF and StoreManageIF provided by *StoreInventory*. These interfaces consist of the methods of class *Store* that are called by the interface objects of *OrderDesk* and *InventoryDesk*, respectively.

The store object in component *StoreSale* and the one in *StoreInventory* are the same object all the time. We compose by union these components to obtain the component *StoreServer* with provided interface StoreIF:

$$StoreServer \mathrel{\widehat{=}} StoreSale \parallel StoreInventory$$

This component provides interface StoreSaleIF to component *SalesHandler*, and interfaces StoreOrderIF and StoreManageIF to *InventoryHandler*. The resulting component-based architecture is shown as a UML *component diagram* in Fig. 16.

With this model of the component-based architecture, we can transform the design sequence diagram of a use case into *component interaction* (or *sequence*) *diagrams*. For example, the inner loop during normal mode from the design sequence diagram in Fig. 14 is transformed into the component sequence diagram in Fig. 17. We call this model of component-based architecture the *logical architecture model* because all the interfaces are still *object-oriented interfaces*, meaning that interactions are realized by method invocations via object references.

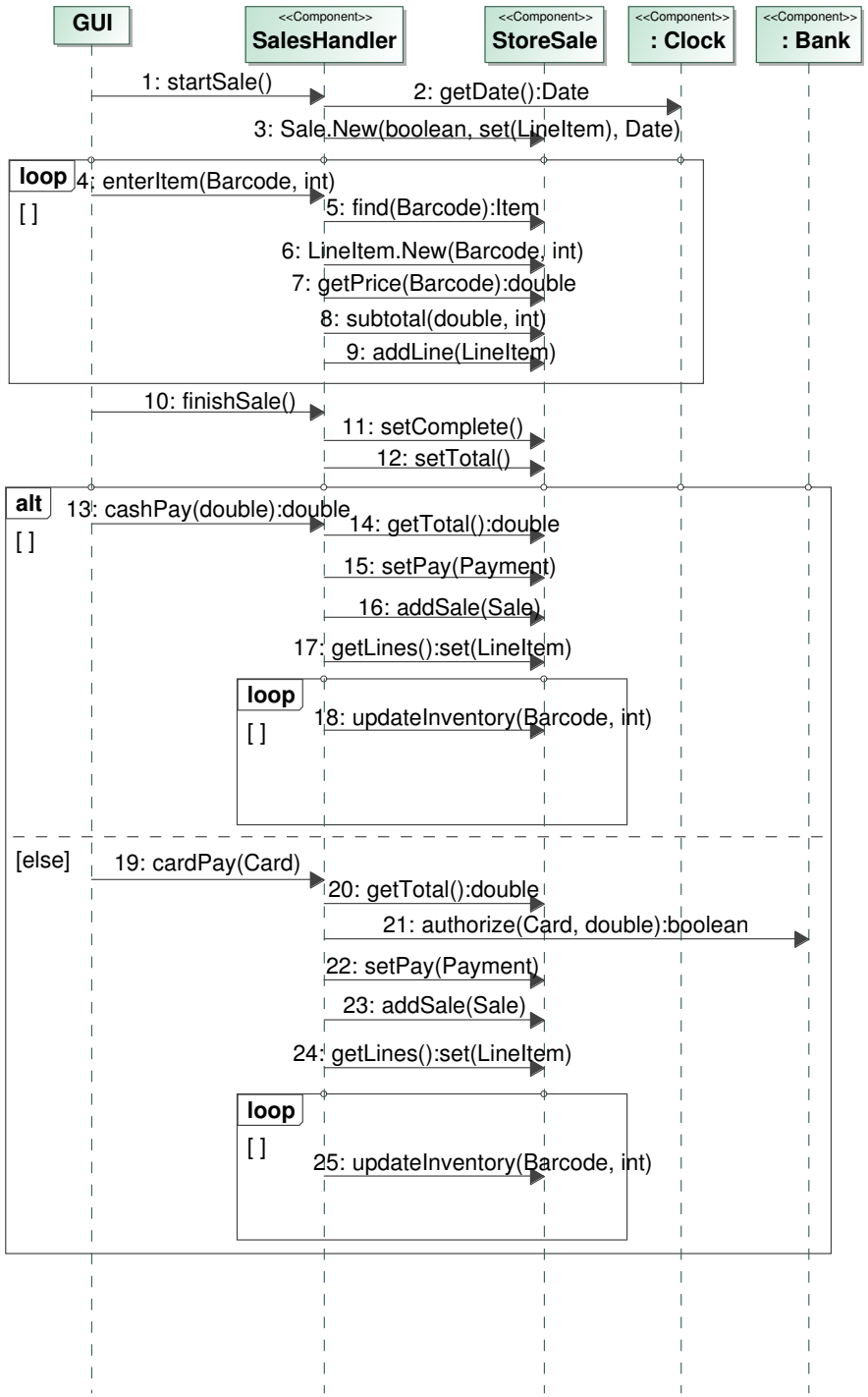


Fig. 17. Component Sequence Diagram

Discussion about the architecture design From the way we construct the architecture model, we can see the importance of the object-oriented design by refinement for the use case methods. A detailed enough design allows us to identify the components and their interfaces.

4.5 Detailed design of CoCoME

In the detailed design, we carry out the refinement activities described in the following paragraphs.

Refining and Refactoring the design of components We refine the object-oriented design of components by refining the bodies of the methods and data encapsulation as described in Subsection 3.2. This is generally called *refactoring* [19]. While data encapsulation can be automated with tool support, refining a method body requires the inspection of the specification of the designs of the methods. The main purpose of such a decomposition is to improve the understandability and efficiency of the designs. We do not have a formal definition of the matrices of these features, but focus on the assurance of the functional correctness of such a decomposition once a decision is made.

Refactoring is a special case of refinement that usually changes a piece of code to another one with an equivalent behaviour (determinism). Simple refactoring, such as removing method calls from expressions, is useful as well as complicated refactoring. For example, we can change the design of *CashDesk::updateInventory*() to the "wrong design" in **Example** 2, but we add the missing associations to make it a correct design.

Platform specific architecture design In this phase of design, we apply standard designs for individual components and interfaces.

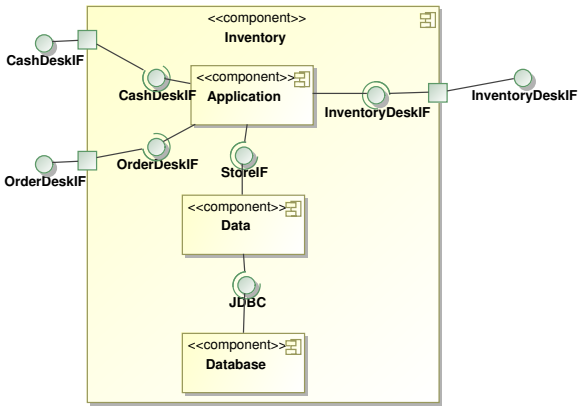


Fig. 18. Inventory decomposition

In the CoCoME application, we use the typical three-layer architecture (*application layer*, *data representation layer* and *database*). The *application layer* is formed by the composition of components *SalesHandler* and *InventoryHandler*, and *StoreServer* the *data layer* that is further decomposed into a component *DataRepresentation* and a *Database* component. The three layer architecture is shown in Fig. 18. All the data are stored in the *Database* and the component *DataRepresentation* is the data representation, that provides the interface StoreIF to the application layer.

When there are many different kinds of data, component *Data* can be further decomposed into components for different kinds of data, such as *SaleData* and *OrderData*.

So far the semantics of the interfaces between the components are still object-oriented interfaces. This works if there is only one *SalesHandler* instance and all the components share a central memory. Now we assume there is more than one *SalesHandler* instance, each having its own clock and only one *StoreServer* instance. We must change some of the object-oriented interfaces by introducing *connectors* or middlewares to realize the interfaces properly:

- We keep the interface StoreIF between the application layer and the data representation layer in *StoreServer* as an object-oriented interface.
- *DataRepresentation* and the *Database* interact through JDBC.
- As all *SalesHandler* instances share the same inventory, we can introduce a connector by which the instances get product information or request the inventory to update a product by passing a product code.
- The interaction between the *SalesHandler* instances and the bank or the product supplier can be made via RMI or CORBA.

Verification of the correctness of the use of the standard interaction protocols and middlewares require their formalization in rCOS. This is not done yet. However, the translation of the object-oriented interfaces into their implementations in these standards is easy as the standards are established with the purpose of realizing object-oriented interactions on different platforms.

4.6 Verification and analysis of the design

Static checking and run time verification can be applied to the components after refactoring and significant algorithm design. Since the specification of functionality design of each component is purely object-oriented and in the style of Eiffel [38] and JML [34], we carry out static checking and testing in the similar way. We translate each rCOS class C into two JML files. One is $C.jml$ that contains the specification translated from the rCOS specification. The other is a Java source file $C.java$ containing the implementation of the specification. During the translation,

the variables used in the rCOS specification are taken as specification-only variables in *C.jml*, and are mapped to program variables in *C.java*.

The translated JML files can be compiled by the JML Runtime Assertion Checker Compiler (*jmlc*). Then, test cases can be executed to check the implementation against the specification. The automatic unit testing tool of JML (*jmlunit*) can generate the unit testing code, and the testing process can be executed with *JUnit*.

```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theCatalog.contains(o);
  @                ((Item)o).theBarcode.equals(c));
  @   assignable theLine, theSale;
  @   ensures   theLine != \old(theLine) &&
  @                theLine.theBarcode.equals(c) &&
  @                theLine.theQuantity == quantity &&
  @                (\exists Object o; theStore.theCatalog.contains(o);
  @                ((Item)o).theBarcode.equals(c) ==>
  @                theLine.theSubtotal == ((Item)o).thePrice * qty) &&
  @                theSale.theLines.size() == (\old(theSale.theLines.size()) + 1) &&
  @                theSale.theLines.contains(theLine);
  @ also
  @ public exceptional_behaviour
  @           requires !(\exists Object o; theStore. theCatalog.contains(o);
  @                           ((Item)o).theBarcode.equals(c));
  @           signals_only Exception;
  @*/
public void enterItem(Barcode c, int qty) throws Exception;
```

Fig. 19. Refined specification of *enterItem*.

For example, the design of *enterItem*() given in Section 4.3 is translated to the *.jml* file shown in Fig. 19. Notice that the text in the dotted rectangle gives the specification for falsifying the precondition. If the unit testing for the specification in Fig. 19 is applied to an implementation that does not throw an exception if the input bar code *c* does not exist, there would be a *NormalPostconditionError* reported by *JUnit*. This indicates that the implementation does not handle the input that falsifies the precondition. We now modify the implementation to the code given in the left side of Fig. 20. An exception will be thrown if the variable *t* is *false*, which represents the nonexistence of the bar code in the catalog. However, with this implementation, an *InvariantError* will be reported. The unsatisfied *invariant* is given in Fig. 20, asserting that each bar code of a sale's line item must have a product with that code in the catalog. Debugging is required to make sure that the bar code must be checked before a line is created and added to the sale. The corrected code is shown on the right of Fig. 20.

Testing is of course not sufficient for correctness. Therefore, it is also desirable to carry out static analysis, for instance with ESC/Java [8].

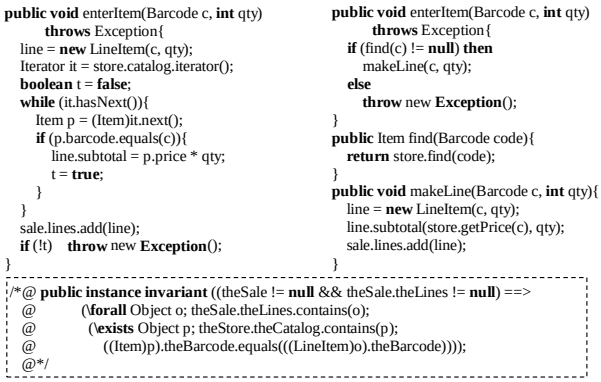


Fig. 20. Improved code of *enterItem*.

4.7 Tool support

In previous work, we investigated the possibility of using Software Engineering- or Model-Driven Development tools like MasterCraft [52,11] or ModelMorf [56,57] to facilitate the rCOS process in existing tools. Clearly, extending an industrial-strength tool is beyond the capabilities of a research project. Using the QVT (Query/View/Transformation)[42] model transformation language implemented in ModelMorf to define refinement steps revealed that it is very inconvenient from a purely syntactical perspective to handle input models corresponding to rCOS programs, especially the functionality specifications.

Current efforts concentrate on harnessing existing tools to give the developer a broad range of analyses to run on a project: multi-view consistency of the different diagrams (Sequence-, State-, and Class Diagrams) and the functionality specifications considers both the static aspect such as well-formedness and correct use of methods, but also dynamic aspects, like making sure that the Sequence- and the State Diagrams are trace equivalent. To do this, translation of the dynamic specification into CSP is automated, although it requires additional annotations to the model.

Also, automatic translation from rCOS to Java code is ongoing, very much in the same way as presented in the paper. Additionally, JML annotations will be emitted for the code.

The input for the above automation is an rCOS use case defined in the UML 2.0 metamodel [43] using Class-, Collaboration-, and Component Diagrams together with a separate profile that uses stereotypes to associate the entities to the different steps in the rCOS development process. That way, we are able to treat stereotyped UML models exported in XMI-format [44] from applications like MagicDraw [41], or created from within our own Eclipse-plugin using the Eclipse Graphical Modelling framework and TOPCASED [53].

5 Conclusions and Related Work

We have only presented a small part of the whole case study in CoCoME. Our experience in working with the case study shows that multi-view modelling and separation of concerns are helpful in the modelling and design. And, for correctness analysis and correct design a semantic model is needed to relate the models of the different views.

We have introduced the semantic theory of rCOS and with the study shown how it is used to formalize the concepts of object- and component-based systems in a model driven development. In particular, we showed that a model driven development process can be easily adapted to use rCOS concepts. In particular, we have identified the clear relation between the development activities, concepts and artifacts and their formalisation in rCOS, and possible tool support.

Properties are specified in rCOS by logical formulas, and in analysis of these, algebraic properties of modelling elements are used. The algebraic properties form the foundation for model transformations. To ensure consistency and correctness, both static and dynamic consistency of the specification must be checked, and both abstraction and refinement techniques are needed for model transformation and analysis.

The work also shows that different models and tools are more effective for the design and analysis of some aspects than others. Proved correct model transformations should be carried out side by side with verification and validation. rCOS is a methodology that supports consistent use of different techniques and tools for modelling, design, verification and validation.

Related formalisms

Model based formalisms like VDM [29], B [50] and Z [54] have been used extensively in software specification and verification. They are designed for modular specification and compositions in the procedural programming paradigm and are effective at modelling data structures as sets and relations between sets. Unlike rCOS, they do not define concepts like components, interfaces and objects as first class elements. They do not have semantics dealing with pointers or references and therefore they, and even their object-oriented extensions VDM++ [17] and Object-Z [51], do not support sophisticated object-oriented mechanisms of object-oriented programming languages, such as dynamic binding and polymorphism.

Eiffel [38] first introduced the idea of design by contracts for object-oriented programming. The notion of designs for methods in object-oriented rCOS is similar to the use of assertions in Eiffel, and thus also supports similar techniques for static analysis and testing. JML [34] has recently become a popular language for modelling and analysis of object-oriented designs. It shares similar ideas of using assertions and refinement as behavioral subtypes in Eiffel. The strong point of JML is that it is

well integrated with Java and comes with parsers and tools for runtime checking and testing. Other similar languages and techniques include ESC/Java [8] and Spec♯ [4]. The notion of contracts in both Eiffel and JML specifies the change of the states of the objects by an execution step of a method. In this sense, they are object-oriented counterparts of VDM and Z. The main difference of rCOS from these techniques is that the model of contracts is a multi-view model, and it supports the specification of component-based architectures, that include interaction protocols and dynamic behavior.

In Fractal [46], behavior protocols are used to specify interaction behavior of a component. rCOS also uses traces of method invocations and returns to model the interaction protocol of a component with its environment. However, in rCOS the protocol does not have to be a regular language. Also, for components, rCOS separates the protocol of the provided interface methods from that of the required interface methods. This allows better pluggability among components. On the other hand, the behavior protocols of components in Fractal are the same for the protocols of coordinators and glue units that are modelled as processes in rCOS. In addition to interaction protocols, rCOS also supports state-based modelling with guards and pre-post conditions. This allows us to carry out stepwise functionality refinement; but at the cost of decidability and thus fully automated checking.

We share many ideas with work done at York University by the group of Woodcock on *Circus* [7], the work on TCOZ of Dong at SNU [37], and the work at Oldenburg by the group of Olderog on linking CSP-OZ with UML [45]. In these approaches, multi-notational modelling languages are used to encompass different views of a system. However, rCOS emphasises on semantic unification that has taken UTP as its single point of departure and thus avoids some of the complexities of merging existing notations. Yet, the CSP-OZ framework has the virtue of well-developed underlying frameworks and tools, that is inspiring to our current work in the development on tool support. Perhaps, a clearer advantage of rCOS is its direct formulation of the engineering concepts and artifacts in component-based model driven design, and the smooth link between component-based and object-oriented modelling and design.

Acknowledgements

We would like to thank the anonymous referees for their very constructive and detailed comments that helped us a lot to bring this paper into this form. This work is supported in parts by the projects HighQSoftD and HTTS funded by the Macau Science and Technology Fund, NSFC-60673114 and 863 of China 2006AA01Z165. The first author is partially supported by the National Basic Research Program of China (973) under Grant No.2005CB321802 and NSFC under Grant No.90612009. We would like to thank our colleagues who have made contributions in the development of rCOS and CoCoME, He Jifeng, Xiaoshan Li, Charles Morisset, E-Y Kang, Jing Liu, Chen Xin, Zhao Liang, Lu Yang, and Joseph Okika.

References

[1] R.-J. Back, L. Petre, I. P. Paltor, Formalising UML use cases in the refinement calculus, Tech. Rep. TUCS-TR-279, Turku Centre for Computer Sciennce and Åbo Akademi University, Finland (May 1999).

[2] R.-J. Back, J. von Wright, Trace refinement of action systems, in: 5th International Conference on Concurrency Theory (CONCUR '94), vol. 836 of Lecture Notes in Computer Science, Springer, 1994.

[3] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Graduate Texts in Computer Science, Springer, 1998.

[4] M. Barnett, K. M.Leino, W. Schulte, The Spec# Programming System: An Overview, in: Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04), vol. 3362 of Lecture Notes in Computer Science, Springer, 2005.

[5] P. Borba, A. Sampaio, M. Cornélio, A refinment algebra for object-oriented programming, in: L. Cardelli (ed.), Proc. of ECOOP03, vol. 2743 of Lecture Notes in Computer Science, Springer, 2003, pp. 457–482.

[6] A. Cavalcanti, D. Naumann, A weakest precondition semantics for an object-oriented language of refinement, in: World Congress on Formal Methods, vol. 1709 of Lecture Notes in Computer Science, Springer, 1999, pp. 1439–1460.

[7] A. Cavalcanti, A. Sampaio, J. Woodcock, A refinement strategy for Circus, Formal Aspects of Computing 15 (2-3) (2003) 146–181.

[8] P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond assertions: Advanced specification and verification with JML and ESC/Java2, in: Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, vol. 4111 of Lecture Notes in Computer Science, Springer, 2006.

[9] K. Chandy, J. Misra, Parallel Program Design: a Foundation, Addison-Wesley, 1988.

[10] X. Chen, J. He, Z. Liu, N. Zhan, A model of component-based programing, in: F. Arbab, M. Sirjani (eds.), Intl. Symp. on Fundamentals of Software Engineering (FSEN07), vol. 4767 of Lecture Notes in Computer Science, Springer, 2007.

[11] X. Chen, Z. Liu, V. Mencl, Separation of concerns and consistent integration in requirements modelling, in: Proc. Current Trends in Theory and Practice of Computer Science (SOFSEM07), vol. 4362 of Lecture Notes in Computer Science, Springer, 2007.

[12] Z. Chen, A. Hannousse, D. Hung, I. Knoll, X. Li, Z. Liu, Y. Liu, Q. Nan, J. Okika, A. Ravn, V. Stolz, L. Yang, N. Zhan, Modelling with relational calculus of object and component systems - rCOS, Tech. Rep. 382, UNU/IIST, P.O. Box 3058, Macao, to appear in Lecture Notes in Computer Science (2007).

[13] Z. Chen, X. Li, Z. Liu, V. Stolz, L. Yang, Harnessing rCOS for tool support - the CoCoME experience, in: Z. L. Cliff Jones, J. Woodcock (eds.), Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays, vol. 4700 of Lecture Notes in Computer Science, Springer, 2007, pp. 83–114.

[14] Z. Chen, Z. Liu, A. Ravn, V. Stolz, N. Zhan, Refinement and verification in component-based model driven design, Tech. Rep. 388, UNU/IIST, P.O. Box 3058, Macao (2007).

[15] Z. Chen, Z. Liu, V. Stolz, The rCOS tool, VDM-Overture workshop at FM 2008, to be published as Technical Report (2008).

[16] The Concurrency Workbench.
URL http://homepages.inf.ed.ac.uk/perdita/cwb/

[17] E. Dürr, E. Dusink, The role of VDM^{++} in the development of a real-time tracking and tracing system, in: J. Woodcock, P. Larsen (eds.), Proc. of FME'93, vol. 670 of Lecture Notes in Computer Science, Springer, 1993.

[18] C. Flanagan, et al., Extended Static Checking for Java, in: Proc. of the ACM SIGPLAN 2002 Conf. on Programming language design and implementation (PLDI'02), ACM, 2002.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[20] E. Gamma, et al., Design Patterns, Addison-Wesley, 1995.

[21] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison Wesley, 1996.

[22] J. He, X. Li, Z. Liu, Component-Based Software Engineering, in: D. V. Hung, M. Wirsing (eds.), Proc. 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2005), vol. 3722 of Lecture Notes in Computer Science, Springer, 2005.

[23] J. He, X. Li, Z. Liu, rCOS: A refinement calculus for object systems, Theoretical Computer Science 365 (1-2) (2006) 109–142.

[24] J. He, X. Li, Z. Liu, A theory of reactive components, in: Z. Liu, L. Barbosa (eds.), Intl. Workshop on Formal Aspects of Component Software (FACS 2005), vol. 160 of ENTCS, Elsevier, 2006.

[25] C. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[26] C. Hoare, Verified software: Theories, tools, experiments, in: B. Meyer, J. Woodcock (eds.), VSTTE Conference, Verified Software: Theories, Tools, Experiments, vol. 4171 of Lecture Notes in Computer Science, Springer, 2007, pp. 21–29.

[27] C. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.

[28] G. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003.

[29] C. Jones, Systematic Software Development using VDM (2nd Edition), Prentice Hall, 1990.

[30] P. Kruchten, The Rational Unified Process—An Introduction, Addison-Wesley, 2000.

[31] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, 2002.

[32] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd ed., Prentice-Hall International, 2001.

[33] K. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, STTT 1 (1-2) (1997) 134–152.

[34] G. Leavens, JML's rich, inherited specification for behavioural subtypes, in: Z. Liu, J. He (eds.), Proc. 8th Intl. Conf. on Formal Engineering Methods (ICFEM'06), vol. 4260 of Lecture Notes in Computer Science, Springer, 2006.

[35] X. Li, Z. Liu, Prototyping system requirements model, ENTCS 207 (2008) 17–32.

[36] Z. Liu, V. Mencl, A. P. Ravn, L. Yang, Harnessing theories for tool support, to Appear in Proc. International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA06), IEEE Computer Society. Full version as UNU-IIST Technical Report 343, http://www.iist.unu.edu (2006).

[37] B. Mahony, J. Dong, Deep semantic links of TCSP and Object-Z: TCOZ approach, Formal Aspects of Computing 3 (2) (2002) 146–160.

[38] B. Meyer, Eiffel: The Language, Prentice Hall, 1992.

[39] R. Milner, A Calculus of Communicating Systems, Springer, 1980.

[40] C. Morgan, Programming from Specifications, 2nd ed., Prentice Hall, 1994.

[41] NoMagic, Inc., MagicDraw.
URL http://www.magicdraw.com/

[42] Object Management Group, MOF QVT final adopted specification, ptc/05-11-01, http://www.omg.org/docs/ptc/05-11-01.pdf (2005).

[43] Object Management Group, Unified Modeling Language: Superstructure, version 2.0, final adopted specification (2005).
URL http://www.omg.org/cgi-bin/doc?formal/05-07-04

[44] Object Management Group, XML Metadata Interchange (2005).
URL http://www.omg.org/cgi-bin/doc?formal/2005-09-01

[45] E.-R. Olderog, H. Wehrheim, Specification and (property) inheritance in CSP-OZ, Science of Computer Programming 55 (2005) 227–257.

[46] F. Plasil, S. Visnosky, Behavior protocols for software components, IEEE Trans. Software Eng. 28 (11) (2002) 1056–1070.

[47] R. Reussner, et al., CoCoME - the common component modelling example, to appear in Lecture Notes in Computer Science (2008).

[48] A. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1997.

[49] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, 1999.

[50] A. Schneider, The B-method, Masson, 2001.

[51] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.

[52] Tata Consultancy Services, Mastercraft, http://www.tata-mastercraft.com.

[53] Topcased—Open Source Engineering Workshop, http://topcased.org.

[54] J. Woodcock, J. Davies, Using Z: Specification, Refinement, and Proof, Prentice Hall, 1996.

[55] J. Woodcock, C. Morgan, Refinement of state-based concurrent systems., in: Proc. of VDM Europe'90, vol. 428 of Lecture Notes in Computer Science, Springer, 1990.

[56] L. Yang, V. Mencl, V. Stolz, Z. Liu, Automating correctness preserving model-to-model transformation in MDA, in: Proc. of Asian Working Conference on Verified Software, UNU-IIST Technical Report 348, 2006.

[57] L. Yang, V. Stolz, Integrating refinement into software development tools, ENTCS 207 (2008) 69–88.

[58] L. Zhao, X. Liu, Z. Liu, Z. Qiu, Graph transformations for object-oriented refinement, Formal Aspects of Computing, Springer, Published online: 8 January 2008.